高等学校计算机基础教育教材精选

Visual C# 大学程序设计

崔舒宁 杨振平 贾应智 杨琦 编著

清华大学出版社

高等学校计算机基础教育教材精选

Visual C♯ 大学程序设计

崔舒宁　杨振平　贾应智　杨　琦　编著

清华大学出版社
北　京

内 容 简 介

本书以 Visual Studio 2013 为平台，讲述了关于 C# 的编程知识。全书共分为 14 章，其中第 1～10 章主要讲述控制台下的 C#，讲述的重点为面向对象的编程思想；第 11～13 章讲述 Windows 窗体程序的设计，介绍常用的控件、GDI+以及文件读写等方面的知识；最后一章简单讲述常用的数据结构，如线性表和栈等。

本书从基础讲起，是 C# 的入门书籍，可作为高等学校程序设计课程的教材，也可供 C# 程序设计爱好者自学使用。

图书在版编目(CIP)数据

Visual C# 大学程序设计/崔舒宁等编著. —北京：清华大学出版社，2016
高等学校计算机基础教育教材精选
ISBN 978-7-302-42304-1

Ⅰ. ①V… Ⅱ. ①崔… Ⅲ. ①C 语言—程序设计—高等学校—教材 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2016)第 287007 号

责任编辑：焦 虹
封面设计：何凤霞
责任校对：焦丽丽
责任印制：杨 艳

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 **邮 编**：100084
社 总 机：010-62770175 **邮 购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn，010-62795954
印 刷 者：三河市君旺印务有限公司
装 订 者：三河市新茂装订有限公司
经 销：全国新华书店
开 本：185mm×260mm **印 张**：21.75 **字 数**：499 千字
版 次：2016 年 2 月第 1 版 **印 次**：2016 年 2 月第 1 次印刷
印 数：1～2000
定 价：39.50 元

产品编号：064923-01

出版说明

在教育部关于高等学校计算机基础教育三层次方案的指导下，我国高等学校的计算机基础教育事业蓬勃发展。经过多年的教学改革与实践，全国很多学校在计算机基础教育这一领域中积累了大量宝贵的经验，取得了许多可喜的成果。

随着科教兴国战略的实施以及社会信息化进程的加快，目前我国的高等教育事业正面临着新的发展机遇，但同时也必须面对新的挑战。这些都对高等学校的计算机基础教育提出了更高的要求。为了适应教学改革的需要，进一步推动我国高等学校计算机基础教育事业的发展，我们在全国各高等学校精心挖掘和遴选了一批经过教学实践检验的优秀的教学成果，编辑出版了这套教材。教材的选题范围涵盖了计算机基础教育的三个层次，包括面向各高校开设的计算机必修课、选修课，以及与各类专业相结合的计算机课程。

为了保证出版质量，同时更好地适应教学需求，本套教材将采取开放的体系和滚动出版的方式（即成熟一本、出版一本，并保持不断更新），坚持宁缺毋滥的原则，力求反映我国高等学校计算机基础教育的最新成果，使本套丛书无论在技术质量上还是文字质量上均成为真正的“精选”。

清华大学出版社一直致力于计算机教育用书的出版工作，在计算机基础教育领域出版了许多优秀的教材。本套教材的出版将进一步丰富和扩大我社在这一领域的选题范围、层次和深度，以适应高校计算机基础教育课程层次化、多样化的趋势，从而更好地满足各学校由于条件、师资和生源水平、专业领域等的差异而产生的不同需求。我们热切期望全国广大教师能够积极参与到本套丛书的编写工作中来，把自己的教学成果与全国的同行们分享；同时也欢迎广大读者对本套教材提出宝贵意见，以便我们改进工作，为读者提供更好的服务。

我们的电子邮件地址是：jiaoh@tup.tsinghua.edu.cn。联系人：焦虹。

清华大学出版社

前言

近年来，教育部在国家层面上组织实施了一系列计划，其目的就是为了促进高校探索创新型人才培养的新模式，促进高校建立以问题和课题为核心的教学模式，倡导以学生为主体的本科人才培养和研究型教学，调动学生学习的积极性、创造性和主动性，激发学生的创新思维和创新意识，同时在项目实践中逐步掌握提出问题、解决问题的方法，提高其创新实践能力。

本书以 Visual Studio 2013 为平台，讲述关于 C# 的编程知识。全书共分为 14 章，其中第 1～10 章主要讲述控制台下的 C#，讲述的重点为面向对象的编程思想；第 11～13 章讲述 Windows 窗体程序的设计，介绍常用的控件、GDI+ 以及文件读写等方面的知识；最后一章简单讲述常用的数据结构，如线性表和栈等。

完成本书的讲授需要 48 课时（24 课时授课＋24 课时上机）至 72 课时（36 课时授课＋36 课时上机）。本书只需要学生具备基本的计算机基础知识。在已经学习过一门语言的情况下，讲授第 1、2、4 章的内容时需注意 C# 和其他语言的区别。由于 C# 是纯面向对象的语言，所以本书第 3 章简单介绍了面向对象的知识，使学生对程序的总体架构有一个认识。

第 5 章和第 6 章讲述数组和方法。第 7 章到第 9 章讨论了面向对象的 3 个关键的特性：封装、继承和多态。接口不是本书的重点内容，讲述较为简略。第 10 章讨论了 C# 的异常处理机制。

第 11 章到第 13 章讲述 Windows 的窗体编程。对于控件或组件，建议重点讲述三四个控件，其余的可以安排学生自学。

第 14 章简单讲述常用的数据结构。这些数据结构如线性表、栈和队列等在 C# 中均有对应的类。在了解数据结构的基础上，应该尽量使用这里已有的类来完成任务。第 14 章的内容可以安排在最后讲述，也可以在第 10 章后直接讲述。

本书所有代码在 Windows 7 以及 Visual Studio 2013 Team 下调试通过。本书对版本没有特殊要求，使用 Visual Studio Express 可以完全胜任本书的学习。

参加本书编写的人员有崔舒宁（第 1、2、11～14 章）、杨琦（第 3、4 章）、杨振平（第 5、6、8 章）、贾应智（第 7、9、10 章）。全书由崔舒宁统稿。本书讲义经过在西安交通大学部分班级试用，学生反映良好。由于本书涉及面较广，时间仓促，加之作者水平有限，书中错误和疏漏之处在所难免，恳望各位读者不吝指正。联系邮箱：cuishuning@gmail.com。

崔舒宁

目　录

第1章 C#和.NET

1.1 C#概述

编程工具与消费类电子设备(如移动电话和 PDA)的出现带来了新的问题与需求。因为新版本的共享组件与旧软件不兼容,所以集成不同语言的软件组件很困难,安装问题也很常见。人们需要基于 Web 的程序,以便访问和使用 Internet。随着移动电子设备的普及,客户不再局限于桌面计算机,软件需要让任何人通过各种不同类型的设备访问。

为了满足这些需求,2000 年微软公司推出了 C#编程语言。C#是由微软公司 Anders Hejlsberg 和 Scott Wiltamuth 领导的小组开发的,作为.NET 平台上的语言,C#可以方便地集成到.NET 中。C#源于 C、C++ 和 Java,采它们之所长并增加了自己的新特性。C#是面向对象的,包含强大的预建组件类库,从而使程序员可以迅速地开发程序。C#和 Visual Basic 共享框架类库,Visual C#是事件驱动的可视化编程语言,程序在集成开发环境(IDE)中创建。编写的程序响应定时器和用户启动的事件(如鼠标单击与键击)。除了编写程序语句建立 C#程序之外,还可以用 Visual Studio 的图形用户界面方便地把按钮、文本框之类的预定义对象拖放到屏幕上某个位置,然后标注和缩放它们。Visual Studio 会产生大部分 GUI 代码,利用 IDE,程序员可以方便地生成、运行、测试和调试 C#程序,从而减少生成可工作程序所需的时间,比不用 IDE 要快得多。

1.2 .NET 框架和公共语言运行时

1.2.1 .NET 框架

技术人员一般将微软公司看成平台厂商。微软公司搭建技术平台,而技术人员在这个技术平台上创建应用系统。从这个角度看,.NET 是微软公司的新一代技术平台,可为敏捷商务构建互联互通的应用系统,这些系统是基于标准的、联通的、适应变化的、稳定的和高性能的。从技术的角度看,一个.NET 应用是一个运行于.NET Framework 之上的应用程序。更精确地说,.NET 应用使用.NET Framework 类库编写,并运行于公共语言运行时(Common Language Runtime)之上的应用程序。如果一个应用程序与.NET Framework 无关,就不能叫做.NET 程序。比如,仅使用 XML 就不是.NET 应用,仅使用 SOAP SDK 调用 Web Service 也不是.NET 应用。目前 .NET Framework 的最新版

本为 4.5,和 Visual Studio 2013 一起发布。

1.2.2 CLR 与 CLI

CLI(Common Language Infrastructure,公共语言基层结构)是为虚拟机环境而制定的规范,使得由各种高级语言所编制的程序可以在不同的系统环境中执行而不必更改或重新编译源程序代码。CLI 指定一种标准的虚拟机中介语言,然后将各种高级语言编制的源代码映射为该中间语言。对于.NET Framework 而言,这种中介语言是 Microsoft Intermediate Language(MSIL)。

在中介语言中,执行程序时,代码通过实时编译器(Just-In-Time(JIT)Compiler)最终被映射为机器码。当然,在 CLI 中介语言中,代码可在其他任何具有 CLI 功能的环境中执行。一个具体的环境就是 CLR。

CLR(Common Language Runtime,公共语言运行时)是一个执行程序的标准化环境。不管程序是用 Visual Basic、C#,还是 C++ 等高级语言来编制,都可以在此环境中执行。因此,按照 CLR 语言标准编写的 C++ 程序属于 C++ /CLI,也就是为 CLI 而编制的 C++ 。

CLI 也被定义为一种数据类型的公共集,叫做公共类型系统(Common Type System,CTS)。它可用于任意语言编写的程序,以实现 CLI 为目标。CTS 规定了在 CLR 环境中如何使用数据类型,并包括一些预定义类型。用户可以定义自己的数据类型,但必须用一种特殊方法来定义以便与 CLR 兼容。有一种标准的表达数据类型的系统,允许不同语言编制的组件用统一的方法来处理数据,从而使得不同语言编制的组件可以结合到单个程序中。CLR 不过是 CLI 规范在个人计算机、Windows 操作系统中的执行而已。

毫无疑问,在其他操作系统环境和硬件平台上,CLI 也同样可行。有时会发现术语 CLI 和 CLR 可交换使用,尽管很明显它们不是一回事。CLI 是一种标准规范,而 CLR 是微软公司对 CLI 的实现。

1.2.3 C#的执行过程

执行 C#代码的过程包括下列步骤。

1. 选择编译器

为利用 CLR 环境提供的优点,必须使用一个或多个针对运行时的语言编译器,如 Visual Basic、C#、Visual C++ 、F#或第三方编译器(如 Eiffel、Perl 或 COBOL 编译器)。

2. 将代码编译为 MSIL

当编译为托管(CLR)代码时,编译器将源代码翻译为 Microsoft 中间语言(MSIL),这是一组可以有效地转换为本机代码且独立于 CPU 的指令。MSIL 包括用于加载、存储和初始化对象以及对对象调用方法的指令,还包括用于算术和逻辑运算、控制流、直接内存访问、异常处理和其他操作的指令。要使代码可运行,必须先将 MSIL 转换为特定于 CPU 的代码,这通常是通过实时(JIT)编译器来完成的。由于公共语言运行时为它支持

的每种计算机结构都提供了一种或多种 JIT 编译器，因此同一组 MSIL 可以在所支持的任何结构上编译和运行。

当编译器产生 MSIL 时，它也产生元数据。元数据描述代码中的类型，包括每种类型的定义、每种类型的成员的签名、代码引用的成员和运行时在执行时使用的其他数据。MSIL 和元数据包含在一个可移植、可执行的文件中，文件格式包含 MSIL 或本机代码以及元数据，使得操作系统能够识别公共语言运行时映像。

3. 将 MSIL 编译为本机代码

运行 MSIL 之前，必须先根据公共语言运行时将其编译为适合目标计算机体系结构的本机代码。.NET Framework 提供了两种方式来执行此类转换：.NET Framework 实时(JIT)编译器和.NET Framework Ngen.exe(本机映像生成器)。

应用程序运行时，JIT 编译器可以在加载和执行程序集内容的过程中根据需要将 MSIL 转换为本机代码。由于公共语言运行时为所支持的每种 CPU 体系结构都提供了一个 JIT 编译器，因此开发人员可以生成一组在具有不同的计算机体系结构的不同计算机中进行 JIT 编译和运行的 MSIL 程序集。但是，如果托管代码调用特定于平台的本机 API 或特定于平台的类库，则将只能在该操作系统上运行。

JIT 编译器考虑了在执行过程中某些代码永远不会被调用的可能性。它不是耗费时间和内存将文件中的所有 MSIL 都转换为本机代码，而是在执行期间根据需要转换 MSIL 并将生成的本机代码存储在内存中，供该进程上下文中的后续调用访问。在加载并初始化类型时，加载程序将创建存根(stub)并将其附加到该类型的每个方法中。首次调用某个方法时，存根会将控制权交给 JIT 编译器，后者会将该方法的 MSIL 转换为本机代码，并修改存根以使其直接指向生成的本机代码。这样，对 JIT 编译的方法的后续调用将直接转到该本机代码。

由于 JIT 编译器会在调用程序集中定义单个方法时将该程序集的 MSIL 转换为本机代码，因而必定会对运行时的性能产生不利影响。在大多数情况下，这种性能降低是可以接受的。更为重要的是，由 JIT 编译器生成的代码会绑定到触发编译的进程上，它无法在多个进程之间共享。为了能在多个应用程序调用或共享一组程序集的多个进程之间共享生成的代码，公共语言运行时支持一种提前编译模式。此提前编译模式使用 Ngen.exe(本机映像生成器)将 MSIL 程序集转换为本机代码，其作用与 JIT 编译器极为相似。但是，Ngen.exe 的操作与 JIT 编译器的操作有 3 点不同：

(1) 它在应用程序运行之前而不是在应用程序运行过程中执行从 MSIL 到本机代码的转换。

(2) 它一次编译一个程序集，而不是一次编译一个方法。

(3) 它将本机映像缓存中生成的代码以文件的形式持久保存在磁盘上。

在编译为本机代码的过程中，MSIL 代码必须通过验证过程，除非管理员已经建立了允许代码跳过验证的安全策略。验证过程检查 MSIL 和元数据以确定代码是否是类型安全的，这意味着它仅访问已授权访问的内存位置。类型安全帮助将对象彼此隔离，因而可以保护它们免遭无意或恶意的破坏。它还提供了对代码可靠地强制进行安全限制的

保证。

验证过程中检查 MSIL 代码，确认该代码只能通过正确定义的类型访问内存位置和调用方法。例如，代码不允许以超出内存范围的方式来访问对象。另外，验证过程检查代码以确定 MSIL 是否已正确生成，这是因为不正确的 MSIL 会导致违反类型安全规则。验证过程通过正确定义的类型安全代码集，并且它只通过类型安全的代码。然而，由于验证过程存在一些限制，某些类型安全代码可能无法通过验证，而某些语言在设计上并不产生可验证的类型安全代码。如果安全策略要求提供类型安全代码，而该代码不能通过验证，则在运行该代码时将引发异常。

4. 运行代码

公共语言运行时提供使托管代码执行能够发生以及可在执行期间使用的各种服务的基础结构。在运行方法之前，必须先将其编译为特定于处理器的代码，调用已经为其生成 MSIL 的每个方法；运行该方法时，该方法将是 JIT 编译的。下次运行该方法时，将运行现有的 JIT 编译的本机代码。这种进行 JIT 编译然后运行代码的过程一直重复到执行完成时为止。在执行过程中，托管代码接收若干服务，这些服务涉及垃圾回收、安全性、与非托管代码的互操作性、跨语言调试支持、增强的部署以及版本控制支持等。

1.2.4 垃圾回收

在 C++ 编程中，需要自己来管理申请内存和释放内存，于是有了 new、delete 关键字，还有一些内存申请和释放函数。C++ 程序必须很好地管理自己的内存，不然就会造成内存泄漏(Memory Leak)。在.NET 时代，微软公司为开发人员提供了强有力的机制——垃圾回收(Garbage Collection，GC)。垃圾回收机制是 CLR 的一部分。我们不用操心内存何时释放，从而可以花更多精力关注应用程序的业务逻辑。CLR 里的垃圾回收机制可用一定的算法判断某些内存程序不再使用，回收这些内存并交给系统使用。

1.3 面向对象的编程技术

面向对象的程序设计(Object Oriented Programming，OOP)是软件开发人员多年来从真实世界的建模中受到启发所创造的一种软件开发的方法。利用面向对象的编程方法可以创建功能异常复杂的软件，利用其中的代码复用可以加速软件开发的过程，并增强软件的后期功能的可扩展性与可维护性。

在面向对象的编程方法诞生之前，软件开发人员采用面向过程的程序设计语言编程。面向过程的程序设计语言最主要的特征是采用子过程(子模块)进行软件开发，但随着现代软件的功能越来越强大，面向过程的编程方法明显无法适应大型软件的开发及维护，因此诞生了面向对象的编程方法。

1.3.1 类和对象

在面向对象的编程方法中，类与对象是其核心。对象代表一个存在的实体(实际物

体)或者实际的概念,对象由类建模而成。例如,对雇员、窗口、汽车等都可以建立相应的模型。

面向对象的程序设计思想是现代软件开发的基础。面向对象的程序由各种各样的类、对象和方法组成,它们有机地组织在一起从而实现了复杂的软件功能设计。在面向对象的程序设计中,类与对象是核心,而对象需要由类建模得到。我们在程序中将使用对象来完成不同的功能。对象是消息、数据和行为的组合,对象可以接收和发送消息并使用消息进行交互。消息包含要传递给接收对象的信息。

对象由类得到,类转换为对象的过程称之为类的实例化。因此对象就是类所表现出来的一个实实在在的例子,一个"具体的东西"或者"具体的事情"。对象可以是有形的实体(实际物体),也可以是无形的概念。对象往往有边界,有属性,有方法(特定用途)。例如,张三这个人:

(1) 是有形的实体,有边界。

(2) 有姓名、身高、年龄、性别等,有属性。

(3) 可以走、跑、跳、叫、唱等,有方法。

一般而言,对象应具有以下特性:

- 对象有状态,对象的状态由对象的各种属性和相应的值构成。
- 对象可以显示行为(即方法),对象的行为(方法)使对象可以完成相应的功能。
- 对象有唯一的身份,对象的身份可以把它与其他对象区别开来。

例如,一辆汽车,它有状态,即它可能正在行驶或者已经静止。它有方法,可以左转、右转、减速、加速等。它有一个车牌号(身份),唯一地标识了这辆汽车。两个对象可能什么都相同,但它们不可能有相同的身份,例如相同的两辆轿车,它们肯定会有不同的车牌号。

在当今的现实生活中到处都是形状、颜色各异,大小、功能不同的对象,所以现实生活就是由各种各样的对象构成的。为了对世界上各种各样的对象进行研究,了解和掌握这些对象的习性、功能和使用方法,科学家将世界上的各种对象进行了逻辑上的分类,例如矿物类、植物类、动物类等;并继续将这些类进行更细的逻辑上的分类,例如动物类分成了鸟类、哺乳动物类、鱼类等。每一个正在运动的哺乳动物、正在飞行的鸟、正在水里游动的鱼都是实实在在的对象,科学家怎样将它们划分到哺乳动物类、鸟类或者鱼类中呢?毫无疑问,科学家根据它们所表现出来的共同特征将这些对象进行了分类。

例如,孔雀、麻雀和翠鸟都属于鸟类,因为它们具有鸟类家族的公共特性,即它们都产蛋,都覆盖羽毛,都有空的骨架结构且都能飞翔。所以,我们可以将类理解成对对象进行分类的一个模型,而该类中的对象都是由这个模型所产生的,它们具有这个类模型的共同特征。同样,现实生活中的《C#编程技术》和《ASP.NET 编程技术》都是"书"这个类的实际对象,都是由"书"这个类模型所得到的对象。它们拥有"书"这个类的共同特征。

1.3.2 字段

属性在类的声明中(参见例 1-1)用变量表示,这样的变量称为字段(field),它是在类

声明的内部声明的，但是位于类的方法声明体之外。当类的每个对象维护自己的属性副本时，代表属性的字段又称为实例变量。类的每个对象（实例）在内存中有该变量的一个单独实例。

字段和方法（见 1.3.3 节）都具有访问权限。通常有 3 种访问权限：private、public 和 protected。以 private 修饰的字段或者方法只允许在类的内部使用，也就是说只允许类中的其他方法调用或者访问。以 public 修饰的字段或者方法则没有什么限制，从类的外部可以直接使用。protected 修饰的字段和方法可以在派生类（见第 8 章）中使用。

1.3.3 方法

方法（Method）描述了类实际执行任务的机制，从形式上看，相当于 C 语言的函数。类中的方法，可以直接存取类中的字段的值。C＃语言被称为纯面向对象的语言。也就是说，所有的方法都在某个类中，不存在独立于类的方法。

例 1-1 一个简单类的示例。

```
1: public class Hello
2: {
3:     private int x=1;                //定义 x为整数,其值为 1
4:     public void DisplayMessage()
5:     {
6:         Console.WriteLine("Hello");
7:     }
8: }
```

程序第 1 行，声明了一个类 Hello。在第 3 行，声明了一个 private 的字段，含有一个整型变量 x。程序的第 4 行到第 8 行，声明了一个 public 的方法 DisplayMessage，该方法在控制台输出 Hello 字样。

类不能直接使用，如前所述，需要根据类来创建对象，也就是类的实例化。语句如下：

```
Hello myHello=new Hello();
```

上述语句通过 new 关键字，创建了 Hello 类的一个对象实例 myHello，随后可以通过语句

```
myHello.DisplayMessage();
```

来调用 Hello 类中的方法。

1.3.4 注释

在编写程序的过程中，明了清晰的注释是至关重要的。一般一段复杂的代码经过一段时间后，即使是自己当时绞尽脑汁想出的，可能也会忘得一干二净。如果当时做了注释，就能帮助我们快速回忆起编程思路和需求背景的内容，可以快速投入修改和功能添加等工作中。此外，注释也可以帮助别人快速理解代码，便于交流。C＃注释一般常用的有以下 3 种。

(1) 单行注释：较为短小的注释，以//开头直到本行结束，用于对一行代码的注释。如例 1-1 中的第 3 行。

(2) 多行注释：标注类、方法、参数、返回值的内容等，以///开头。如果有多行，则每一行都以///开头。在方法或类的前面输入///后，Visual Studio 具有自动完成功能，会补齐其余的部分。

(3) 长段注释：描述需求、版本信息等。多行大段的注释以/＊开头，并且以＊/结尾。

1.4　使用 Visual Studio

在 C＃编程中，程序可以分为两大类：一类是带有 Windows 窗体的标准 Windows 程序，这类程序将在本书的后半部分介绍；一类是控制台程序，本书前几章都将使用控制台程序，其程序的输入和输出是文本，位于控制台窗口。在 Windows 系统中，控制台窗口被称为命令提示符。

下面通过示例讲解如何使用 Visual Studio 2013 创建一个完整的 C＃控制台程序。

例 1-2　控制台程序示例。在本例中，完善例 1-1，使其可以运行。

(1) 启动 Visual Studio 2013，如图 1-1 所示。选择“新建项目”(也可以在菜单中选择“新建项目”)。

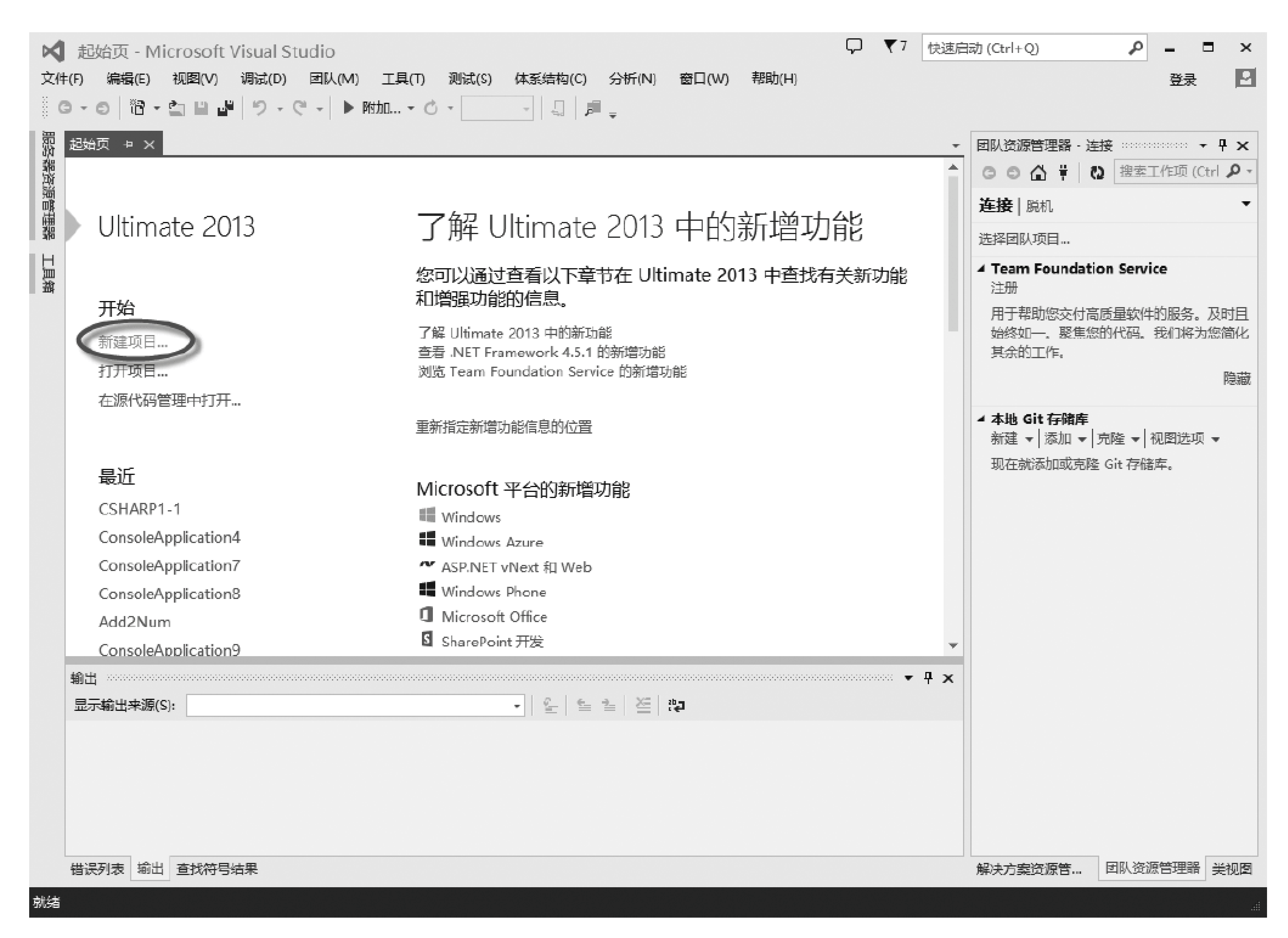


图 1-1　新建一个项目

（2）在弹出的对话框中，选择 Visual C＃→“控制台应用程序”，并给出一个合适的名称，然后单击“确定”按钮，如图 1-2 所示。

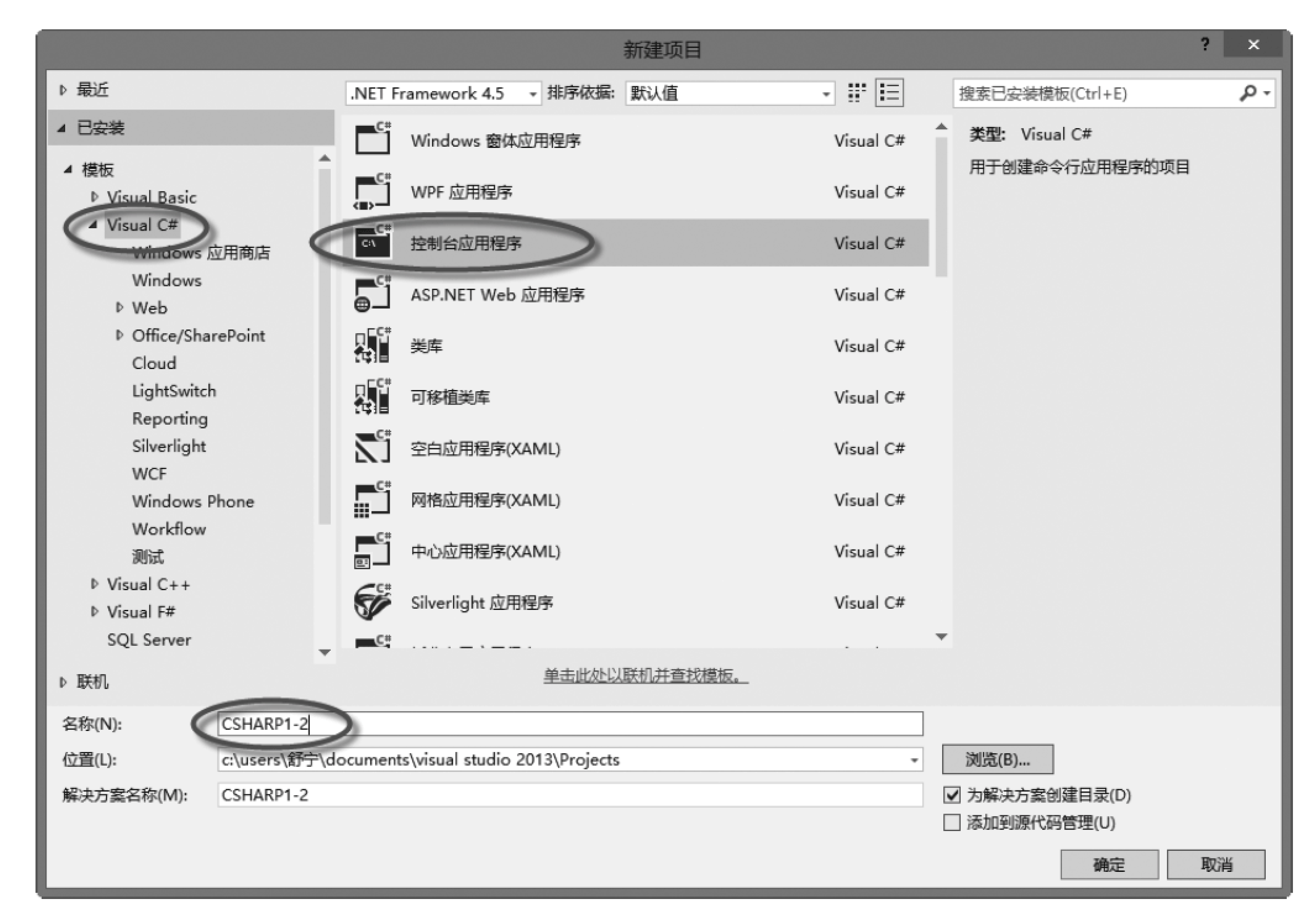


图 1-2　建立 Visual C＃控制台应用程序

（3）在随后出现的窗口中，Visual Studio 创建了文件 Program. cs，并在里面添加了一些代码，如图 1-3 所示。

（4）补充输入完整的程序，以下是程序的全部代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSHARP1_2
{
    class Program
    {
        static void Main(string[] args)
        {
            Hello myHello=new Hello();
            myHello.DisplayMessage();
        }
    }
```

```
17:     public class Hello
18:     {
19:         private int x=1;
20:         public void DisplayMessage()
21:         {
22:             Console.WriteLine("Hello");
23:         }
24:     }
25: }
```

程序第 1～5 行是对名字空间(见 1.5 节)的引用。第 7 行将本项目的所有元素都组织到了名字空间 CSHARP1_2 中,该名字空间是在创建项目时使用的项目名称。在本项目中含有两个类,其中第 17～24 行是例 1-1 的内容。在 class Program 中含有方法 Main。对于每一个 C#程序,至少应该含有一个 Main 方法,它是整个程序执行的起点。同时注意到 Main 方法的前面有一个 static 的修饰符。该修饰符会在后面详细讨论,此处需要知道的是:含有 static 修饰的方法,不需要实例化,可以直接执行。

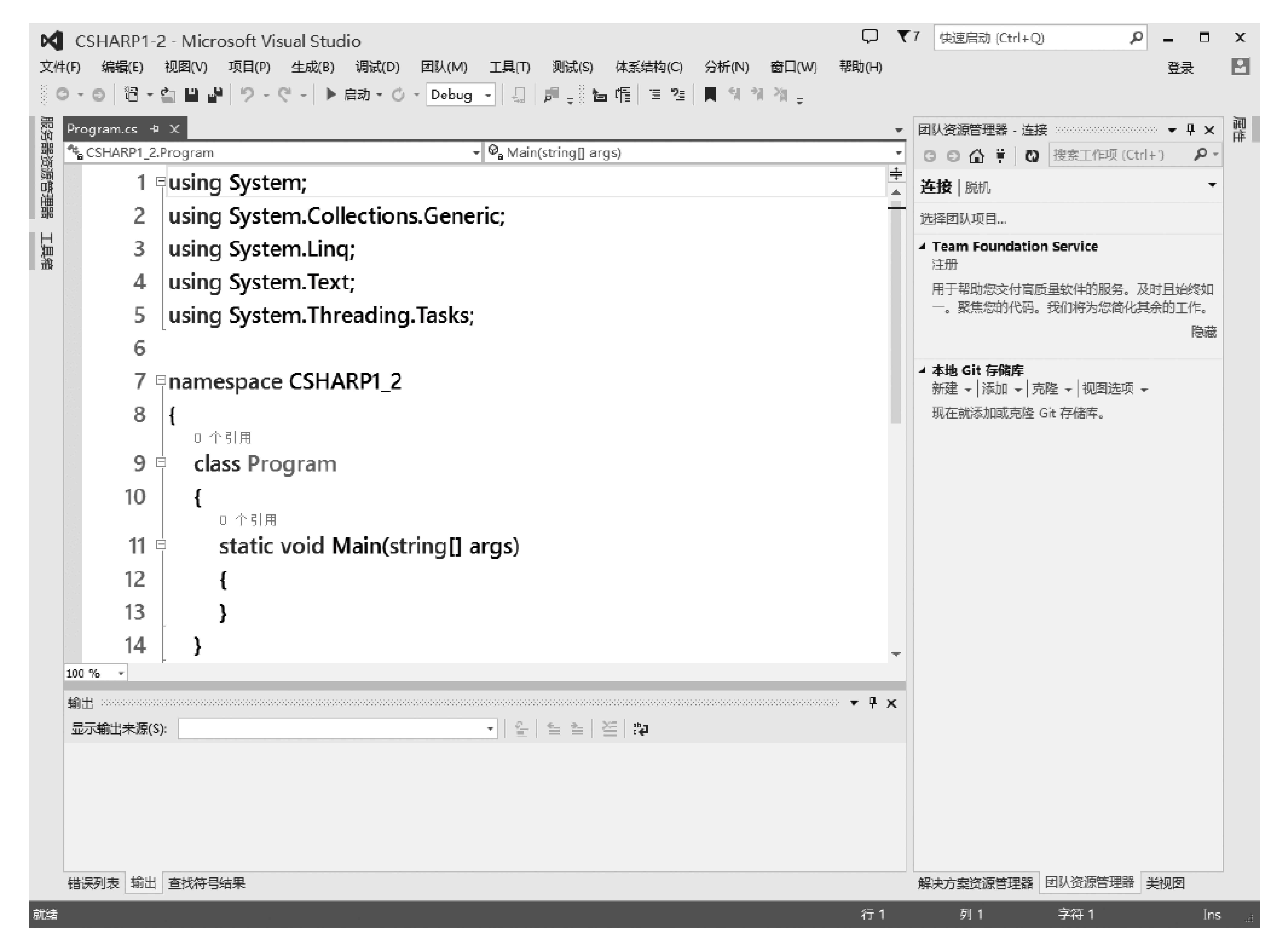


图 1-3　在此处输入所需的代码,编写程序

在 Main 中,实例化了一个 Hello 对象,并调用了 DisplayMessage 方法。编写好程序后,按下 Ctrl+F5 观看程序的运行结果。

1.5 名字空间

namespace 中文多翻译为名字空间(或者命名空间),之所以使用它,是因为可用的单词太少,并且不同的人写的程序不可能所有的变量(类名、标识符)都没有重名现象,对于库来说(不同的人写的代码汇集到库中)这个问题尤其严重。如果两个人写的库文件中出现同名的变量,使用起来就有问题了。为了解决这个问题,引入了名字空间这个概念,使用 namespace,所使用的标识符就是在该名字空间中定义的。这样一来就不会引起不必要的冲突了。名字空间是用来组织和重用代码的编译单元。

所谓 namespace,是指标识符的各种可见范围。在例 1-1 的 DisplayMessage 中使用了一个 WriteLine 方法,该方法是 Console 类的一个静态(static)方法,所以可以直接使用,而无须实例化,类似于例 1-2 中的 Main 方法。Console 类是组织在 System 名字空间中的,所以要完整地使用该方法应该是:

```
System.Console.WriteLine("Hello");
```

很多时候写复杂的全名不太方便,于是有了 using 语句。在名字的开头使用 using 语句表示引用该名字空间,在代码书写中可以省略名字空间。在例 1-2 中,因为有

```
using System;
```

所以后面的 Console. WriteLine 语句省略了前面的 System。

namespace 是 C# 的一个关键字。实际上,它只起标识作用,把需要的内容放到一起细化管理。也可以定义自己的名字空间,如:

```
namespace SpaceName
{
    //declaration
}
```

在 C# 中每一个新建的工程,都会自动放入到一个新的名字空间中(例 1-2 第 7 行)。在例 1-2 中前面引入了许多名字空间,这是编译器为了方便编程自动加的。对于例 1-2 来讲,只需要对 System 名字空间引用即可。

1.6 解决方案和项目

通俗地理解,一个项目可以就是所开发的一个软件。项目可以表现为多种类型,如控制台应用程序、Windows 应用程序、类库(Class Library)、Web 应用程序、Web Service、Windows 控件等。如果经过编译,从扩展名来看,应用程序都会被编译为 exe 文件,而其余的会被编译为 dll 文件。既然是 exe 文件,就表明它是可以被执行的。表现在程序中,这些应用程序都有一个主程序入口点,即 Main()。而类库,如 Windows 控件等,则没有这个入口点,所以也不能直接执行,而仅提供一些功能给其他项目调用。

在 Visual Studio 中,可以在"文件"菜单中,选择"新建"一个"项目",来创建一个新的项目。例如创建控制台应用程序。注意在此时,Visual Studio 除了建立了一个控制台项目之外,该项目同时还属于一个解决方案。这个解决方案有什么用？如果只需要开发一个 Hello World 的项目,解决方案自然毫无用处。但是,一个稍微复杂一点的软件都需要很多模块来组成,为了体现彼此之间的层次关系,利于程序的复用,往往需要多个项目。每个项目实现不同的功能,最后将这些项目组合起来,就形成了一个完整的解决方案。形象地说,解决方案就是一个容器,在这个容器里,分成好多层、好多格,用来存放不同的项目。一个解决方案与项目是大于等于的关系。建立解决方案后,会建立一个扩展名为 sln 的文件。

在解决方案里添加项目,不能再用"新建"的方法,而是要在"文件"菜单中,选择"添加"。添加的项目,可以是新项目,也可以是已经存在的项目。

对于本课程的学习,需要注意的是,一般而言,每一道题目应该对应一个解决方案,而不是一个项目。也就是说,在初期的练习阶段,每做一道题目应该从一个新的解决方案开始。同样,要打开一个曾经写过的程序,不应该直接打开 C# 的源文件,而应该打开解决方案,即双击 sln 文件。

1.7 控制台的输入与输出

一个实际的程序,总是离不开输入和输出的。严格地讲,输入和输出并不是 C# 语言的一部分。输入和输出依靠预先提供的方法来完成。这些方法都包含在 Console 类中。Console 类中的方法都是静态的,因此不需要创建一个 Console 类的对象(实例)。

所谓输入和输出对于本书的例子而言,输入是指在程序运行时,从键盘得到要处理的信息。输出是指将结果显示在屏幕上。在此,只讨论字符界面(控制台的输入和输出)。

控制台是一个操作系统窗口,用户可在其中通过计算机键盘输入文本,并从计算机终端读取文本输出,从而与操作系统或基于文本的控制台应用程序进行交互。Console 类对从控制台读取字符并向控制台写入字符的应用程序提供基本支持。Console 类提供用于从控制台读取单个字符或整行的方法;该类还提供若干写入方法,可将值类型的实例、字符数组以及对象集自动转换为格式化或未格式化的字符串,然后将该字符串写入控制台。

需要注意的是,实际上输入和输出都是针对字符和字符串(或文本)的。因此,输出到控制台时,所有数据都应该转换为字符或字符串。然而,很多时候这种转换是自动的,我们并不需要提供额外的代码来转换。而输入则没有这种自动转换,从键盘的输入将作为一个字符或字符串被读取,可能需要额外的代码将其转换为整数、小数或真正需要的数据类型。

1.7.1 控制台的输出

控制台的输出主要通过 Console.Write 以及带有换行的 Console.WriteLine 来完成,下列程序段向控制台(屏幕)输出整数、字符串等。

```
int a=56;
Console.Write(a);
Console.Write("Hello World");
```

Console. Write 和 Console. WriteLine 的功能基本是一样的，主要区别在于 WriteLine 输出完之后会跟一个换行符，下一个输出会从一个新行开始。另外，需要注意的是 Console 类中并非只有一个 Write 方法，而是有多个，只不过这些方法是同名的而已。上面的程序段中，输出整数的和输出字符串的实际是两个不同的 Write 方法。这种现象称为重载。

Write 和 WriteLine 方法还有一个重要的用法是使用占位符的方式。例如：

```
Console.Write("X+Y={0}+{1}={2}",X,Y,X+Y);
```

在上面的语句中，最终输出的是 Write 参数的第一个字符串 X＋Y＝{0}＋{1}＝{2}，但是{0}等是占位符，实际输出的时候，{0}、{1}和{2}的值将被 X、Y 和 X＋Y 的值所取代。因此，要求占位符的个数应该和后面替换的参数个数一致且一一对应。

例 1-3 Console. Write 的输出。

```
01: using System;
02:
03: namespace CSHARP1_3
04: {
05:     class Program
06:     {
07:         static void Main(string[] args)
08:         {
09:             int a=23;
10:             int b=34;
11:             Console.Write("Welcome C#  2013");
12:             Console.Write("  ");            //输出 2个空格
13:             Console.Write(a);
14:             Console.Write('\n');            //换行符
15:             Console.Write("A={0},B={1},A+B={2}", a, b, a+b);
16:         }
17:     }
18: }
```

程序第 11 行实现了字符串的输出，第 13 行是变量的输出，第 15 行是占位符的使用。程序的运行结果如下：

```
Welcome C#  2013 23
A=23,B=34,A+B=57
```

除了使用{0}作为占位符之外，还可以加入其他参数，进一步控制输出的格式。例如：

```
decimal i=123.45678;
```

```
Console.WriteLine("{0:C}", i);
Console.WriteLine("{0:F}", i);
Console.WriteLine("{0:E}", i);
```

第一行将按货币格式输出，第二行按定点格式（小数点后 2 位）输出，第三行按科学计数法输出。本书不在此罗列所有对输出格式的控制了，在以后用到的地方再解释。

1.7.2 控制台的输入

控制台的输入大多数时候是指从键盘输入，也就是程序在运行的时候从键盘得到要处理的数据。这主要依靠两个方法 Console. Read 和 Console. ReadLine 来完成。Console. Read 方法读取一个字符，Console. ReadLine 方法读取一个字符串。

需要注意的是：无论程序最终需要什么类型的数据，在输入时，只能将输入作为字符或者字符串读入。读入数据后，程序需要将读入的数据转换为需要的数据类型（如果有必要的话）。由于 Read 方法每次只能读入一个字符，因此实际使用 ReadLine 更多一些。例如，从键盘输入一个字符串到程序中：

```
string s=Console.ReadLine();
```

如果需要输入一个数字呢？例如输入整数或者是小数。这时，仍然将输入的内容作为一个字符串读入，然后有程序将字符串转换为需要的类型。有多种转换的方式，在这里，使用类 Convert 提供的方法来转换。和 Console 类相似，Convert 也提供了一组静态的方法来完成转换，因此使用 Convert 类的方法时也不需要创建对象。常用的 Convert 方法如表 1-1 所示。

表 1-1 常用的 Convert 方法

方法名称	转换后的数据类型	方法名称	转换后的数据类型
Conver. ToBoolean	boolean	Convert. ToSingle	single
Convert. ToInt32	integer	Convert. ToDouble	double
Convert. ToInt64	long	Convert. ToChar	char
Conver. ToDecimal	decimal	Convert. ToDateTime	datetime

例 1-4 从键盘输入两个小数，相加后显示结果。

```
using System;

namespace CSHARP1_4
{
    class Program
    {
        static void Main(string[] args)
        {
            string s=Console.ReadLine();
```

```
                decimal a=Convert.ToDecimal(s);
                s=Console.ReadLine();
                decimal b=Convert.ToDecimal(s);
                decimal c=a+b;
                Console.WriteLine("{0}+{1}={2}",a,b,c);
            }
        }
}
```

程序运行时,先将数据读入到 s 中,再把字符串转换为小数。每次输入一个数时,需要按下回车键,即一行输入一个数据。程序运行的结果如下:

```
23.987
14.67
23.987+14.67=38.657
```

上面的例子中先将 23.4 作为字符串"23.4"存储到了变量 s 中,然后再转换为小数存储到 a 中。要注意需要确保这样的转换是可行的。如果试图将字符串"ABC"转换为小数,程序会立即中止运行并报错。

如果想在一行内输入多个数据,则需要做更多的工作。首先,这些数据会作为 1 个字符串读入,当采用空格来分隔时,形如"12 45 23 56 88"。此时在一行内输入了 5 个整数,并以空格分隔。接下来需要将这个字符串分解为 5 个字符串,存入到一个字符串数组中。分解的依据是空格作为分隔符。可以使用字符串对象自己的一个方法 Split 来完成此事,然后使用循环依次转换。

例 1-5 在一行内输入多个数据。在一行内输入 5 个整数,分别求其除 3 的余数。

```
using System;

namespace CSHARP1_5
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("请输入 5个整数,以空格分隔并以回车结束");
            string s=Console.ReadLine();
            string[] sSplit=s.Split(' ');
            for (int i=0; i<5; i++)
            {
                int a=Convert.ToInt32(sSplit[i]);
                Console.Write((a % 3).ToString()+" ");
            }
            Console.WriteLine();
        }
```

```
19:         }
20: }
```

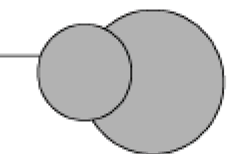
上述程序的第 10 行 ReadLine 将空格分隔的 5 个整数作为一个字符串读入到变量 s 中。随后调用变量 s(也是一个数组对象)的方法 Split 将 s 按空格分隔成 5 个字符串结果存入字符串数组 sSplit 中(程序第 11 行)。在输出的 Write 方法中,a%3 的结果是一个整数。整数也是一个对象,使用 ToString 方法将其转换为一个字符串,在和含有一个空格的字符串连接后输出(程序第 15 行)。假设输入的 5 个整数是 23、45、6、7 和 8,程序的运行结果下:

```
请输入 5个整数,以空格分隔并以回车结束
23 45 6 7 8
2 0 0 1 2
```

习题

1. 建立第一个 C# 程序,输出“我的名字叫……”。
2. 从键盘输入两个整数,分别输出这两个整数和、差、积、商。
3. 在一行内输入 6 个整数,输出最大的一个整数。

第2章



数据类型与表达式

2.1　.NET 数据类型

.NET 中的数据类型的两个基本类别是“值类型”和“引用类型”。基元类型、枚举和结构为值类型。类、字符串、标准模块、接口、数组和委托为引用类型。

所有的数据类型要么是值类型，要么是引用类型，只有一种情况例外——根类型 System.Object。根类型非常特殊，因为它既不是引用类型也不是值类型，而且不能实例化。因此，类型 Object 的变量可包含值类型或引用类型，如图 2-1 所示。

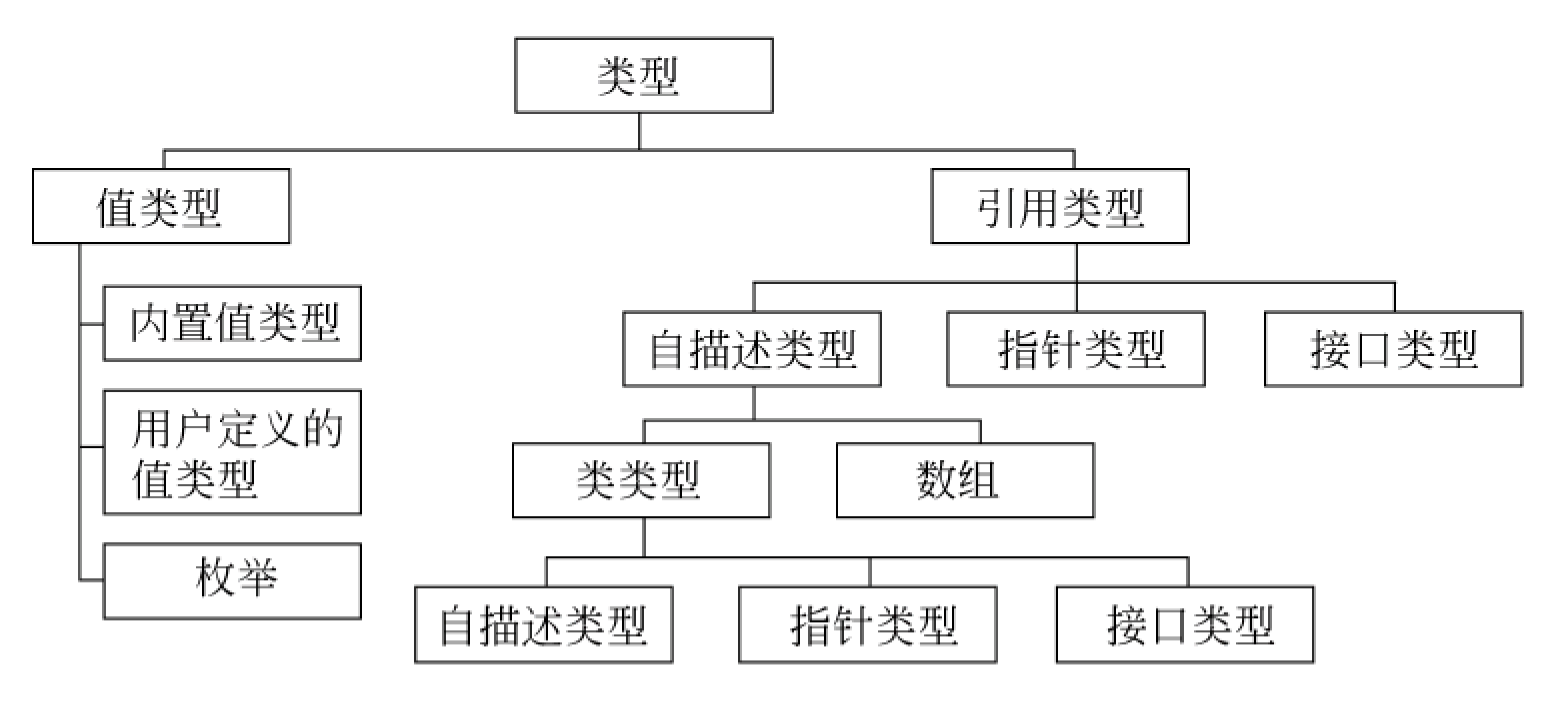


图 2-1　.NET 中的数据描述

2.1.1　值类型

如果数据类型在它自己的内存分配中存储数据，则该数据类型就是“值类型”。值类型包括：

- 所有数字数据类型。
- Boolean、Char 和 Date。
- 所有结构，即使其成员是引用类型。
- 枚举，因为其基础类型总是 SByte、Short、Integer、Long、Byte、UShort、UInteger 或 Ulong。

2.1.2　引用类型

引用类型包含指向存储数据的其他内存位置的指针。引用类型包括：

- String。
- 所有数组,即使其元素是值类型。
- 类类型,如 Form。
- 委托。

值类型(Value Type)实例通常分配在线程的堆栈(Stack)上,并且不包含任何指向实例数据的指针,因为变量本身就包含了其实例数据。值类型直接包含其数据。值类型的实例要么在堆栈上,要么内联在结构中。引用类型(Reference Type)实例分配在托管堆(Managed Heap)上,变量保存了实例数据的内存引用。引用类型存储对值的内存地址的引用,位于堆上。引用类型可以是自描述类型、指针类型或接口类型,而自描述类型进一步细分成数组和类类型(见图 2-1)。数据在内存中的分配位置,取决于该变量的数据类型。因此,值类型通常分配在线程的堆栈上,而引用类型通常分配在托管堆上,由 GC(垃圾回收)来控制其回收。

2.2　C#的数据类型

C#的数据是.NET 数据在 C#语言上的一个实现。因此,C#同样有两种类型:值类型和引用类型。值类型的变量直接包含其数据,而引用类型的变量存储对其数据的引用,后者称为对象。对于引用类型,两个变量可能引用同一个对象,因此对一个变量的操作可能影响另一个变量所引用的对象。对于值类型,每个变量都有其数据副本(除 ref 和 out 参数变量外),因此对一个变量的操作不可能影响另一个变量。

C#的值类型进一步划分为简单类型(simple type)、枚举类型(enum type)和结构类型(struct type),C#的引用类型进一步划分为类类型(class type)、接口类型(interface type)、数组类型(array type)和委托类型 (delegate type)。表 2-1 为 C#类型系统的概述。

表 2-1　C#类型系统

类　别		说　明
值类型	简单类型	有符号整型:sbyte,short,int,long
		无符号整型:byte,ushort,uint,ulong
		Unicode 字符:char
		IEEE 浮点型:float,double
		高精度小数:decimal
		布尔型:bool
	枚举类型	enum E{...}形式的用户定义的类型
	结构类型	struct S{...}形式的用户定义的类型

续表

类　　别		说　　明
引用类型	类类型	所有其他类型的最终基类：object
		Unicode 字符串：string
		class C{...}形式的用户定义的类型
	接口类型	interface I{...}形式的用户定义的类型
	数组类型	一维和多维数组，例如 int[]和 int[,]
	委托类型	delegate TD(...)形式的用户定义的类型

2.2.1 简单类型

C#共支持8种整型类型，分别是8位、16位、32位和64位整数值的有符号和无符号的形式。两种浮点类型：float和double，分别使用32位单精度和64位双精度的格式表示。decimal类型是128位的数据类型，适合用于财务计算和货币计算。C#的bool类型用于表示布尔值为true或false的值。在C#中字符和字符串处理使用Unicode编码。char类型表示一个16位Unicode编码单元，string类型表示16位Unicode编码单元的序列(注意string是引用类型)。C#的数值类型如表2-2所示。

表 2-2 C#的数值类型

类　　别	位　数	类　型	范围及精度
有符号整型	8	sbyte	－128～127
	16	short	－32768～32767
	32	int	－2147483648～2147483647
	64	long	－9223372036854775808～9223372036854775807
无符号整型	8	byte	0～255
	16	short	0～65535
	32	uint	0～4294967295
	64	ulong	0～18446744073709551615
浮点数	32	float	1.5×10^{-45}～3.4×10^{38}，7位精度
	64	double	5.0×10^{-324}～1.7×10^{308}，15位精度
小数	128	decimal	1.0×10^{-28}～7.9×10^{28}，28位精度

2.2.2 枚举

C#程序使用类型声明(type declaration)创建新类型。类型声明指定新类型的名称和成员。有五种类别的C#类型是可由用户定义的：类类型、结构类型、接口类型、枚举

类型和委托类型。本节讲述枚举类型，下一节讲述结构，其他类型在后续章节学习。

枚举类型是具有命名常量的独特的类型。每种枚举类型都具有一个基础类型，该基础类型必须是 8 种整型之一。没有显式地声明基础类型的枚举声明意味着所对应的基础类型是 int。枚举类型的值集和它的基础类型的值集相同。枚举的声明如下：

```
访问修辞符 enum 枚举名 :基础类型
{
    枚举成员
}
```

枚举成员是该枚举类型的命名常数。任意两个枚举成员不能具有相同的名称。每个枚举成员均具有相关联的常数值。此值的类型就是枚举的基础类型。每个枚举成员的常数值必须在该枚举的基础类型的范围之内。

例 2-1　简单的枚举示例。

```
public enum TimeofDay:uint
{
    Morning=1,
    Afternoon=2,
    Evening=3
}
```

在枚举类型中声明的第一个枚举成员其默认值为零。以后枚举成员的值是将前一个枚举成员的值加 1 而得到。这样增加后的值必须在该基础类型可表示的值的范围内；否则，会出现编译时错误。允许多个枚举成员有相同的值，没有显示赋值的枚举成员的值，总是前一个枚举成员的值+1。

例 2-2　枚举成员的赋值。

```
public enum Number        //没有显示声明基础类型,默认为 int
{
    a=1,
    b,
    c=1,
    d
}
```

在此，b 的值为 2，d 的值为 2。

System.Enum 类型是所有枚举类型的抽象基类，并且从 System.Enum 继承的成员在任何枚举类型中都可用。System.Enum 本身不是枚举类型。相反，它是一个类类型，所有枚举类型都是从它派生的。存在从任何枚举类型到 System.Enum 的装箱转换，并且存在从 System.Enum 到任何枚举类型的拆箱转换(装箱和拆箱见 2.2.4 节)。

例 2-3　一个完整的例子。

```
01: using System;
02:
```

```
namespace CSHARP2_3
{
    public enum TimeofDay
    {
        Morning,
        Afternoon,
        Evening
    }
    class Program
    {
        static void Main(string[] args)
        {
            WriteGreeting(TimeofDay.Morning);
            WriteGreeting(TimeofDay.Evening);
            WriteGreeting(TimeofDay.Afternoon);
        }
        static void WriteGreeting(TimeofDay timeofDay)
        {
            switch (timeofDay)
            {
                case TimeofDay.Morning:
                    Console.WriteLine("goodmorning");
                    break;
                case TimeofDay.Afternoon:
                    Console.WriteLine("goodafternoon");
                    break;
                case TimeofDay.Evening:
                    Console.WriteLine("goodevening");
                    break;
            }
        }
    }
}
```

2.2.3 结构

结构类型与类类型相似，表示一个带有数据成员和函数成员的结构。但是，与类不同，结构是一种值类型，并且不需要堆分配。结构类型不支持用户指定的继承，并且所有结构类型都隐式地从类型 object 继承。

结构是使用 struct 关键字定义的，与类相似，都表示可以包含数据成员和函数成员的数据结构。结构是一种值类型，并且不需要堆分配。所以，结构的实例化可以不使用 new 运算符。在结构声明中，除非字段被声明为 const 或 static，否则无法初始化。结构不能声明默认构造函数（没有参数的构造函数）或析构函数，但可以声明带参数的构造函数（关

于构造函数，参见第 3 章类的构造函数）。结构在赋值时进行复制。将结构赋值给新变量时，将复制所有数据，并且对新副本所做的任何修改不会更改原始副本的数据。结构类型的变量直接包含了该结构的数据，而类类型的变量所包含的只是对相应数据的一个引用（被引用的数据称为“对象”），但是结构仍可以通过 ref 和 out 参数（参见第 6 章）引用方式传递给函数成员。结构可用于允许为 null 的类型，因而可向其赋 null 值。

例 2-4　结构的简单示例。

```
struct A
{
    public int x;                                  //不能直接对其进行赋值
    public int y;
    public static string str=null;                 //静态变量可以初始化
    public A(int x,int y)                          //带参数的构造函数
    {
        this.x=x;
        this.y=y;
        Console.WriteLine("x={0},y={1},str={2}",x,y,str);
    }
}
```

2.2.4　拆箱与装箱

C＃的类型系统是统一的，因此任何类型的值都可以按对象处理。C＃中的每个类型直接或间接地从类型 object 派生，而 object 是所有类型的最终基类。引用类型的值都被当作“对象”来处理，这是因为这些值可以简单地被视为属于 object 类型。值类型的值则通过执行装箱（boxing）和拆箱（unboxing）操作按对象处理。下面的示例将 int 值转换为 object，然后又转换回 int。

```
int i=123;
object o=i;          //装箱
int j=(int)o;        //拆箱
```

将值类型的值转换为类型 object 时，将分配一个对象实例以包含该值，并将值复制到该箱子中。反过来，将一个 object 引用强制转换为值类型时，将检查所引用的对象是否含有正确的值类型，如果是，则将箱子中的值复制出来。C＃的统一类型系统实际上意味着值类型可以“按需”转换为对象。由于这种统一性，object 类型的通用库（例如.NET Framework 中的集合类）既可以用于引用类型，又可以用于值类型。

2.2.5　常量

常量是在编译时已知并在程序的生存期内不发生更改的不可变值。常量使用 const 修饰符进行声明。只有 C＃内置类型（System.Object 除外）可以声明为 const。用户定义的类型（包括类、结构和数组）不能为 const。请使用 readonly 修饰符创建在运行时初始

化一次即不可再更改的类、结构或数组。常量必须在声明时初始化。例如：

```
public const int months=12;
```

在此示例中，常量 months 始终为 12，不可更改，即使是该类自身也不能更改它。实际上，当编译器遇到 C# 源代码（例如 months）中的常量修饰符时，将直接把文本值替换到它生成的中间语言（IL）代码中。因为在运行时没有与常量关联的变量地址，所以 const 字段不能通过引用传递，并且不能在表达式中作为左值出现。可以同时声明多个相同类型的常量，例如：

```
const int months=12, weeks=52, days=365;
```

在书写数值型常数时，一般小数将会被解释为浮点型，而不是 decimal。在程序中书写一个十进制的数值常数时，C# 默认按照如下方法判断一个数值常数据属于哪种 C# 数值类。如果一个数值常数不带小数点，如 12345，则这个常数的类型是整型。对于一个属于整型的数值常数，C# 按如下顺序判断该数的类型：int、uint、long、ulong。如果一个数值常数带小数点，如 3.14，则该常数的类型是浮点型中的 double 类型。

如果不希望 C# 使用上述默认的方式来判断一个十进制数值常数的类型，则可通过给数值常数加后缀的方法来指定数值常数的类型。可以使用的数值常数后缀有以下几种。

- u（或者 U）后缀：加在整型常数后面，代表该常数是 uint 类型或者 ulong 类型。具体是其中的哪一种，由常数的实际值决定。C# 优先匹配 uint 类型。
- l（或者 L）后缀：加在整型常数后面，代表该常数是 long 类型或者 ulong 类型。具体是其中的哪一种，由常数的实际值决定。C# 优先匹配 long 类型。
- ul 后缀：加在整型常数后面，代表该常数是 ulong 类型。
- f（或者 F）后缀：加在任何一种数值常数后面，代表该常数是 float 类型。
- d（或者 D）后缀：加在任何一种数值常数后面，代表该常数是 double 类型。
- m（或者 M）后缀：加在任何一种数值常数后面，代表该常数是 decimal 类型。

2.2.6 字符和字符串

C# 提供的字符类型数据按照国际上公认的标准，采用 Unicode 字符集。一个 Unicode 字符的长度为 16 位（bit），它可以用来表示世界上大部分语言种类。所有 Unicode 字符的集合构成字符类型。字符类型的类型标识符是 char，因此也可称为 char 类型。凡是在单引号中的一个字符，就是一个字符常数，例如：'a'、'p'、'*'、'0'、'8'。

在表示一个字符常数时，单引号内的有效字符数量必须且只能是一个，并且不能是单引号或者反斜杠（\）。为了表示单引号和反斜杠等特殊的字符常数，C# 提供了转义符（和 C 语言相同）。在需要表示这些特殊常数的地方，可以使用这些转义符来代替字符。

一个字符串是被双引号包含的一系列字符。例如，“how are you!” 就是一个字符串。string 类是专门用于对字符串进行操作的，例如：

```
string str1="中国,";
string str2="你好!";
```

```
string str3=str1+str2;      //这相当于 str3="中国,你好!"
char c=str3[0];             //取出 str3的第一个字符,即"中"字
```

C＃支持以下两种形式的字符串常数。

（1）常规字符串常数：放在双引号间的一串字符，就是一个常规字符串常数，如：

```
"this is a test"
```

除了普通的字符，一个字符串常数也能包含一个或多个前面描述的转义符。

（2）逐字字符串常数：逐字字符串常数以@开头，后跟一对双引号，在双引号中放入字符。如：

```
@"计算机"
```

逐字字符串常数同常规字符串常数的区别在于，在逐字字符串常数的双引号中，每个字符都代表其最原始的意义，在逐字字符串常数中没有转义字符。也就是说，逐字字符串常数双引号内的内容在被接受时是不变的，并且可以跨越多行。唯一的例外是，如果要包含双引号（"），就必须在一行中使用两个双引号（""）。

2.2.7　隐式类型

可以赋予局部变量推断类型 var 而不是显式类型。var 关键字指示编译器根据初始化语句右侧的表达式推断变量的类型。推断类型可以是内置类型、匿名类型、用户定义类型或.NET Framework 类库中定义的类型。

从 Visual C＃ 3.0 开始，在方法范围中声明的变量可以具有隐式类型 var。隐式类型就好像已经声明该类型一样，但由编译器确定类型。例如下面的两个 i 声明在功能上是等效的：

```
var i=10;        //隐式声明
int i=10;        //显式声明
```

2.3　表达式

表达式是由运算符将运算对象（如常数、变量和函数等）连接起来的具有合法语义的式子。在 C＃中，由于运算符比较丰富，因而可以构成灵活多样的表达式。这些表达式的应用一方面可以使程序编写得短小简洁，另一方面还可以完成某些在其他高级程序设计语言中较难实现的运算功能。学习表达式时应注意以下几个方面：

（1）运算符的正确书写方法。运算符与通常在数学公式中所见到的符号有很大差别，例如，整除求余（％）、相等（＝＝）、逻辑运算与（＆＆）等。

（2）运算符与运算对象的关系。运算符可以分为单目运算符、双目运算符，甚至还有复合表达式，其中的两个运算符对三个或者更多个运算对象进行操作。

（3）运算符具有优先级和结合方向。如果一个运算对象的两边有不同的运算符，首先执行优先级别较高的运算。如果一个运算对象两边的运算符级别相同，则应按由左向

右的方向顺序处理。如果编程序时对运算符的优先顺序没有把握,可以通过使用括号来明确其运算顺序。

2.3.1 算术运算符和算术表达式

C#的算术运算符有+(加)、-(减)、*(乘)、/(除)、%(整除求余)。其中/为除法运算符。注意:如果除数和被除数均为整型数据,则结果也是整数。例如,5/3的结果为1。%为整除求余运算符。%运算符两侧均应为整型数据,其运算结果为两个运算对象做除法运算的余数。例如5%3的结果为2。不允许两个算术运算符紧挨在一起,也不能像在数学运算式中那样,任意省略乘号,以及用中圆点“.”代替乘号等。如果遇到这些情况,应该使用括号将连续的算术运算符隔开,或者在适当的位置上加上乘法运算符。例如:x*-y应写成x*(-y),(x+y)(x-y)应写成(x+y)*(x-y)。

2.3.2 关系运算符和关系表达式

关系运算符又称为比较运算符,有6种关系运算符:>(大于)、<(小于)、==(等于)、>=(大于等于)、<=(小于等于)、!=(不等于)。

用关系运算符将两个表达式连接起来就构成了关系表达式。关系表达式的值的类型为逻辑数据类型(布尔型),如:

```
x>=3
a+b==c
```

如果比较运算的结果成立,关系表达式取值就为true,否则关系表达式的值为false。注意,算术运算符的优先级高于关系运算符。即:

```
a+b==c
```

等价于

```
(a+b)==c
```

2.3.3 逻辑运算符和逻辑表达式

简单的关系比较是不能满足实际编程需要的,一般还需要用逻辑运算符将关系表达式或逻辑量连接起来,构成较复杂的逻辑表达式。逻辑表达式的值也是逻辑量。C#中提供了3种逻辑运算符:!(逻辑非)、&&(逻辑与)、||(逻辑或)。

在逻辑运算符中,逻辑与&&的优先级高于逻辑或||的优先级,而所有的关系运算符的优先级均高于以上两个逻辑运算符。至于逻辑非运算符“!”,由于这是一个单目运算符,所以和其他单目运算符(例如用于作正、负号的+和-)一样,优先级高于包括算术运算符在内的所有双目运算符。例如表达式:

```
x*y>z && x*y<100 || -x*y>0 && !isgreat(z)
```

的运算顺序为:

```
计算 x* y                          //算术运算优先于比较运算
计算 x* y>z                        //比较运算优先于逻辑运算
计算 x* y<100                      //比较运算优先于逻辑与运算
计算 x* y>z && x* y<100            //逻辑与运算优先于逻辑或运算
计算 -x                            //单目运算优先于双目运算
计算 -x* y                         //算术运算优先于比较运算
计算 -x* y>0                       //比较运算优先于逻辑运算
计算 isgreat(z)                    //计算函数值优先于任何运算符
计算 !isgreat(z)                   //单目运算优先于双目运算
计算 -x* y>0 && !isgreat(z)        //逻辑与运算优先于逻辑或运算
计算 x* y>z && x* y<100 || -x* y>0 && !isgreat(z)
```

2.3.4　赋值运算符和赋值表达式

C＃将赋值作为一个运算符处理。赋值运算符为＝，用于构造赋值表达式。赋值表达式的格式为：

```
V=e
```

其中 V 表示变量，e 表示一个表达式。赋值表达式的值等于赋值运算符右边的表达式的值。其实，赋值表达式的价值主要体现在其副作用上，即赋值运算符可以改变作为运算对象的变量 V 的值。赋值表达式的副作用就是将计算出来的表达式 e 的值存入变量 V。和其他表达式一样，赋值表达式也可以作为更复杂的表达式的组成部分。例如：

```
i=j=m* n;
```

由于赋值运算符的优先级较低，并列的赋值运算符之间的结合方向为从右向左，所以上述语句的执行顺序是：首先计算出表达式 m＊n 的值；然后再处理表达式 j＝m＊n，该表达式的值就是 m＊n 的值，该值存入变量 j；最后，处理表达式 i＝j＝m＊n，其值即第一个赋值运算符右面的整个表达式的值，因此也就是 m＊n 的值存入变量 i。

2.3.5　自增运算符和自减运算符

自增运算符＋＋和自减运算符－－也是 C＃程序中常用的运算符。＋＋和－－运算符都是单目运算符，其运算对象常为整型变量或指针变量。这两个运算符既可以放在作为运算对象的变量之前，也可以放在变量之后，但对运算对象的值的影响不同。四种表达式的值分别为：

- i＋＋的值和 i 的值相同。
- i－－的值和 i 的值相同。
- ＋＋i 的值为 i＋1。
- －－i 的值为 i－1。

然而，＋＋和－－这两个运算符真正的价值在于它们和赋值运算符类似，在参加运算的同时还改变了作为运算对象的变量的值。＋＋i 和 i＋＋会使变量 i 的值增加 1；类似地，－－i 和 i－－会使变量 i 的值减少 1。

2.3.6 问号表达式

C#中还提供了一种比较复杂的表达式，即问号表达式，又称条件表达式。问号表达式使用两个运算符“?”和“:”对三个运算对象进行操作，格式为：

```
<表达式 1>?<表达式 2>:<表达式 3>
```

问号表达式的值是这样确定的：如果<表达式 1>的值为 Ture，则问号表达式的值就是<表达式 2>的值；如果<表达式 1>的值等于 False，则问号表达式的值为<表达式 3>的值。利用问号表达式可以简化某些选择结构的编程。例如，分支语句：

```
if(x>y)
    z=x;
else
    z=y;
```

等价于语句：

```
z=x>y ? x : y;
```

2.3.7 位运算符

位运算符共有以下几种。

1. 按位与(&)

两个整型数据中的二进制位做“与”运算。“与”运算的规则为：如果参加运算的两个二进制位均为 1，则结果为 1，否则结果为 0。例如：

```
short x=3,y=5;
```

x 值对应的二进制表示为：00000000 00000011，y 值对应的二进制表示为：00000000 00000101。按位与 x&y 的运算过程为：

```
  00000000 00000011
& 00000000 00000101
-------------------
  00000000 00000001
```

因此 x&y 的结果为二进制数 1，换算成十进制也是 1。

2. 按位或(|)

两个整型数据中的二进制位做“或”运算。“或”运算的规则为：只要参加运算的两个二进制位中有一个为 1，则结果就是 1；只有在参加运算的两个二进制位均为 0 的情况下结果才是 0。

3. 按位异或(^)

两个整型数据中的二进制位做“异或”运算。“异或”运算的规则为：如果参加运算的

两个二进制位不同，则运算结果为 1，否则结果为 0。

4. 按位取反（～）

按位取反是单目运算符，只需一个运算对象。按位取反运算将作为运算对象的整型数据中的二进制位做“求反”运算。“求反”运算的规则很简单：如果原来的二进制位为 1，则运算结果为 0，否则结果为 1，即运算结果和原来的数据相反。

5. 左移位运算符（＜＜）

左移位运算用于将整型数据中的各个二进制位全部左移若干位，并在该数据的右端添加相同个数的 0。

6. 右移位运算符（＞＞）

右移位运算用于将整型数据中的各个二进制位全部右移若干位，并在该数据的左端添加相同个数的 0。

C＃的运算符如表 2-3 所示。

表 2-3　C＃的运算符

类　别	表　达　式	说　　明
基本	x. m	成员访问
	x(…)	方法和委托调用
	x[…]	数组和索引器访问
	x＋＋	后增量
	x－－	后减量
	new T(…)	对象和委托创建
	new T[…]	数组创建
	typeof(T)	获得 T 的 System. Type 对象
	checked(x)	在 checked 上下文中计算表达式
	unchecked(x)	在 unchecked 上下文中计算表达式
一元	＋x	表达式的值相同
	－x	求相反数
	!x	逻辑求反
	～x	按位求反
	＋＋x	前增量
	－－x	前减量
	(T)x	显式地将 x 转换为类型 T

续表

<table>
<tr><th>类　别</th><th>表　达　式</th><th>说　　明</th></tr>
<tr><td rowspan="3">乘除</td><td>x * y</td><td>乘法</td></tr>
<tr><td>x/y</td><td>除法</td></tr>
<tr><td>x%y</td><td>求余</td></tr>
<tr><td rowspan="2">加减</td><td>x+y</td><td>加法、字符串串联、委托组合</td></tr>
<tr><td>x−y</td><td>减法、委托移除</td></tr>
<tr><td rowspan="2">移位</td><td>x<<y</td><td>左移</td></tr>
<tr><td>x>>y</td><td>右移</td></tr>
<tr><td rowspan="6">关系和类型检测</td><td>x<y</td><td>小于</td></tr>
<tr><td>x>y</td><td>大于</td></tr>
<tr><td>x<=y</td><td>小于或等于</td></tr>
<tr><td>x>=y</td><td>大于或等于</td></tr>
<tr><td>x is T</td><td>如果 x 属于 T 类型，则返回 true，否则返回 false</td></tr>
<tr><td>x as T</td><td>返回转换为类型 T 的 x，如果 x 不是 T，则返回 null</td></tr>
<tr><td rowspan="2">相等</td><td>x==y</td><td>等于</td></tr>
<tr><td>x!=y</td><td>不等于</td></tr>
<tr><td>逻辑 AND</td><td>x&y</td><td>整型按位 AND，布尔逻辑 AND</td></tr>
<tr><td>逻辑 XOR</td><td>x^y</td><td>整型按位 XOR，布尔逻辑 XOR</td></tr>
<tr><td>逻辑 OR</td><td>x|y</td><td>整型按位 OR，布尔逻辑 OR</td></tr>
<tr><td>条件 AND</td><td>x&&y</td><td>仅当 x 为 true 才对 y 求值</td></tr>
<tr><td>条件 OR</td><td>x || y</td><td>仅当 x 为 false 才对 y 求值</td></tr>
<tr><td>条件</td><td>x?y:z</td><td>如果 x 为 true，则对 y 求值；如果 x 为 false，则对 z 求值</td></tr>
<tr><td rowspan="2">赋值</td><td>x=y</td><td>赋值</td></tr>
<tr><td>Xop=y</td><td>复合赋值，支持的运算符有：*=　/=　%=　+=　−=　<<=　>>=　&=　^=　|=</td></tr>
</table>

其中，有些运算符表 2-3 中没有提到，如 checked 和 unchecked，以及关系检测 is 和 as 等，这些运算符将在后续章节中学习。

2.3.8　表达式中各运算符的运算顺序

大家知道，四则运算的运算顺序可以归纳为“先乘、除，后加、减”，也就是说乘、除运算的优先级别比加、减运算的优先级别要高。C# 语言中有几十种运算符，仅用一句“先乘、除，后加、减”是无法表示各种运算符之间的优先关系的，因此必须有更严格地确定各运算符优先关系的规则。各优先级的顺序如表 2-4 所示。

表 2-4 C#的优先级顺序

优先级	类　别	运　算　符
1	基本	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
2	单目	+ - ! ~ ++x --x (T)x
3	乘法与除法	* / %
4	加法与减法	+ -
5	移位运算	<< >>
6	关系运算	< > <= >= is
7	条件等	== !=
8	位逻辑与	&
9	位逻辑异或	^
10	位逻辑或	｜
11	条件与	&&
12	条件或	‖
13	条件	?:
14	赋值	= *= /= %= += -= <<= >>= &= ^= ｜=

由表 2-4 可以看出，运算优先级的数字越大，优先级别越低。优先级别最高的是括号。因此如果要改变混合运算中的运算次序，或者对运算次序把握不准时，可以使用括号来明确规定运算的顺序。

2.4 常用数学函数

C# 中的数学函数大多是 Math 类的静态方法。Math 类在名字空间 System 中。例如：

```
double y=Math.Cos(0.5);
```

常用的数学方法见表 2-5。

表 2-5 常用的数学方法

名　称	说　明
E	表示自然对数的底，由常数 e 指定。
PI	表示圆的周长与其直径的比值，通过常数 π 指定。
Abs	返回指定数字的绝对值。
Acos	返回余弦值为指定数字的角度。

续表

名　　称	说　　明
Asin	返回正弦值为指定数字的角度。
Atan	返回正切值为指定数字的角度。
Atan2	返回正切值为两个指定数字的商的角度。
BigMul	生成两个 32 位数字的完整乘积。
Ceiling	返回大于或等于指定数字的最小整数。
Cos	返回指定角度的余弦值。
Cosh	返回指定角度的双曲余弦值。
DivRem	计算两个数字的商，并在输出参数中返回余数。
Equals	确定两个 Object 实例是否相等。
Exp	返回 e 的指定次幂。
Floor	返回小于或等于指定数字的最大整数。
IEEERemainder	返回一指定数字被另一指定数字相除的余数。
Log	返回指定数字的对数。
Log10	返回指定数字以 10 为底的对数。
Max	返回两个数字中较大的一个。
Min	返回两个数字中较小的一个。
Pow	返回指定数字的指定次幂。
Round	将值舍入到最接近的整数或指定的小数位数。
Sign	返回表示数字符号的值。
Sin	返回指定角度的正弦值。
Sinh	返回指定角度的双曲正弦值。
Sqrt	返回指定数字的平方根。
Tan	返回指定角度的正切值。
Tanh	返回指定角度的双曲正切值。
ToString	返回表示当前 Object 的 String。
Truncate	计算一个数字的整数部分。

2.5 例题

例 2-5 根据三边长求三角形面积。

利用海伦公式：$A=\sqrt{s(s-a)(s-b)(s-c)}$，其中 a，b，c 分别为三角形三条边的长

度，$s=\frac{1}{2}(a+b+c)$。

```
//求三角形面积
using System;

namespace CSHARP2_5
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please input a,b,c=");
            string inS=Console.ReadLine();
            string[] inSS=inS.Split(' ');
            double a=Convert.ToDouble(inSS[0]);
            double b=Convert.ToDouble(inSS[1]);
            double c=Convert.ToDouble(inSS[2]);
            var s=(a+b+c)/2;
            var area=Math.Sqrt(s*(s-a)*(s-b)*(s-c));
            Console.WriteLine("area={0}",area);
        }
    }
}
```

输入和输出：

```
3 4 5
area=6
```

为简单起见，程序未考虑对数据的检验，即未检查输入的三边长是否能构成一个三角形。实际上，数据检验是程序的重要组成部分，应给予足够的重视。

例 2-6　输入一个四位无符号整数，反序输出这四位数的四个数字字符。

从输入的无符号整数 n 中依次分解出个位数字、十位数字、百位数字、千位数字并依次存放到变量 c1，c2，c3，c4 中。如将 n%10 的值即个位数字存入 c1 中，将 n/10%10 的值即十位数字存入 c2 中，将 n/100%10 的值即百位数字存入 c3 中，将 n/1000 的值即千位数字存入 c4 中。再将各数字值+'0'转为对应的数字字符。

```
//反序输出 4 位无符号整数的四个数字字符
using System;

namespace CSHARP2_6
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Console.WriteLine("Please input one integer 1000<n<9999");
            uint n=Convert.ToUInt32(Console.ReadLine());
            Console.WriteLine("Before inverse the number is: {0}",n);
            char c1=Convert.ToChar(n%10+'0');          //分离个位数字
            char c2=Convert.ToChar(n/10%10+'0');       //分离十位数字
            char c3=Convert.ToChar(n/100%10+'0');      //分离百位数字
            char c4=Convert.ToChar(n/1000+'0');        //分离千位数字
            Console.WriteLine("After inverse the number is: {0}{1}{2}{3}",
                c1,c2,c3,c4);
        }
    }
}
```

输入和输出：

```
Please input one integer between 1000 and 9999:
1234
Before inverse the number is: 1234
After inverse the number is: 4321
```

例 2-7 求一元二次方程 $ax^2+bx+c=0$ 的根，其中系数 a，b，c 为实数，由键盘输入。

设 $\Delta=b^2-4ac$。当 $\Delta=0$ 时，方程有一个重根；当 $\Delta>0$ 时，方程有 2 个不同的实根；当 $\Delta<0$ 时，方程有 2 个共轭的复根。

```
//解一元二次方程
using System;

namespace CSHARP2_7
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please intput a,b,c=");
            string inS=Console.ReadLine();
            string[] inSS=inS.Split(' ');
            double a=Convert.ToDouble(inSS[0]);
            double b=Convert.ToDouble(inSS[1]);
            double c=Convert.ToDouble(inSS[2]);
            double delta=b*b-4*a*c;
            double p=-b/(2*a);
            double q=Math.Sqrt(Math.Abs(delta))/(2*a);
```

```
            if (delta >= 0)
            {
                Console.WriteLine("x1= {0}",p+q);
                Console.WriteLine("x2= {0}",p - q);
            }
            else
            {
                Console.WriteLine("x1= {0}+ {1}i",p,q);
                Console.WriteLine("x1= {0} - {1}i",p,q);
            }
        }
    }
}
```

输入和输出：

```
Please intput a,b,c= 3 4 5
x1= - 0.666666666666667+ 1.10554159678513i
x1= - 0.666666666666667- 1.10554159678513i
```

例 2-8　温度转换：输入一个华氏温度值，计算并输出对应的摄氏温度值。温度转换的表达式是 C＝5＊(F－32)/9。

```
//温度转换
using System;

namespace CSHARP2_8
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("请输入一个华氏温度：");
            double f= Convert.ToDouble(Console.ReadLine());
            double c= 5.0d/9.0d* (f- 32d);
            Console.WriteLine("对应于华氏温度 {0}的摄氏温度为 {1}",f,c);
        }
    }
}
```

输入和输出：

```
请输入一个华氏温度：100
对应于华氏温度 100的摄氏温度为 37.7777777777778
```

例 2-9　大小写转换：输入一个字符，判断它是否为大写字母；如果是，则将其转换为对应的小写字母输出；否则，不用转换直接输出。

ASCII 表中所有的大写字母 A～Z 是连续排列的，所有的小写字母 a～z 也是连续排列的，但大写字母和小写字母并没有排在一起。因此，如果一个字符是大写字符，就可以通过对其 ASCII 码做如下运算转换为对应的小写字母的 ASCII 码：

小写＝大写－'A'＋'a'

```
//大小写转换
using System;

namespace CSHARP2_9
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("请输入一个字母 ：");
            char ch=Convert.ToChar(Console.ReadLine());
            if(ch>='A' && ch<='Z')
                ch=(char)(ch-'A'+'a');
            Console.WriteLine("将大写转换为小写后,该字母为：{0}",ch);
        }
    }
}
```

输入和输出：

```
请输入一个字母：Q
将大写转换为小写后,该字母为：q
```

例 2-10 判断一个四位的整数是否为回文数。回文数是指由该数各位上数字反序构成的数与原数相同。对于四位整数，可以简单地判断两个条件即千位和个位、百位和十位是否相等，所以要先分解出各位数字。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CSHARP2_10
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("请输入一个四位的整数：");
```

```
                int n=Convert.ToInt32(Console.ReadLine());
                //d1,d2,d3,d4 分别用来表示各位数字
                int d1=n/1000;                          //千位
                int d2=n/100 %10;                       //百位
                int d3=n/10 %10;                        //十位
                int d4=n %10;                           //个位
                if (d1 ==d4 && d2 ==d3)
                    Console.WriteLine("该数是回文数");
                else
                    Console.WriteLine("该数不是回文数");
            }
        }
}
```

输入和输出：

```
请输入一个四位的整数：1234
该数不是回文数
```

习题

1. 编写一个程序，要求完成以下要求：

(1) 提示用户输入任意的三个小数。

(2) 显示这三个小数。

(3) 将这三个小数相加，并显示其结果。

(4) 将结果按四舍五入方法转换成整数并显示。

2. 为例 2-8 添加数据检验部分。给出三边长，检验其是否能构成一个三角形的方法是：检查是否任意两边和均大于第三边。如果检验不合格，则输出信息“Error Data!”。

3. 输入两个角度值 x、y，计算下式的值(C# 中三角函数的输入是弧度)。

$$\frac{\sin(|x|+|y|)}{\sqrt{\cos(|x+y|)}}$$

4. 从键盘输入任意三个整数，然后输出这三个数并计算其平均值。

5. 编写一个程序，将字符串“Love”译成密码，译码方法采用替换加密法。其加密规则是：将原来的字母用字母表中其后面的第 3 个字母的来替换，如字母 c 就用 f 来替换，字母 y 用 b 来替换。提示：分别用 4 个字符变量来存储'L'、'o'、'v'和'e'，利用 ASCII 表中字母的排列关系，按照译码方法对各个变量进行运算后输出即可。

6. 输入一个总的秒数，将该秒数换算为相应的时、分、秒。如输入 3600 秒，则输出结果为 1 小时；输入 3610 秒，则结果为 1 小时 10 秒。通过除法和求余运算完成。

7. 编写程序，定义两个整数，用户通过键盘输入两个整数，程序计算它们的和、差、积、商并输出。

第3章 面向对象的编程1

C＃语言是一种完全的面向对象的程序设计语言。它提出了一个概念：类，它的主要思想是将数据（字段）及处理这些数据的相应函数（方法）封装到类中。类是具有相同或相似性质的对象的抽象，对象的抽象是类，类的具体化就是对象，也就是说类的实例是对象。

3.1 面向对象编程

面向对象编程即面向对象程序设计（Object Oriented Programming，OOP），是一种计算机编程架构。在面向对象编程中，计算机程序由能够起到子程序作用的对象组合而成。其三个主要目标是重用性、灵活性和扩展性。为了实现整体运算，每个对象都能够接收信息、处理数据和向其他对象发送信息。

在面向对象编程中，算法与数据结构作为一个整体，称为对象。任何类的对象都具有一定的属性和操作，也能用数据结构与算法两者合二为一来描述，所以用下面的等式来定义对象和程序：

对象＝（算法＋数据结构），程序＝（对象＋对象＋…）

从上面的等式可以看出，程序就是许多对象在计算机中相继表现自己，而对象则是一个个程序实体。

面向对象编程中的概念主要包括类、对象、数据抽象、数据封装、继承、动态绑定、多态性、消息传递。通过这些概念，面向对象的思想得到了具体的体现。其中最基本的三个特征是封装、继承和多态性，下面简要介绍。

（1）封装。封装是把一个对象的外部特征和内部实现细节分离开来，其他对象可以访问该对象的外部特征，但不能访问其内部实现细节。对象的封装是一种信息隐藏技术，其目的是将对象的使用者与设计者分开。封装隐藏了实现的细节，使得代码模块化。

（2）继承。通过继承可以创建派生类（子类）和基类（父类）之间的层次关系，派生类（子类）可以从其基类（父类）中继承属性和方法。通过继承可以实现代码的重用，从已存在的类派生出的一个新类将自动具有原来那个类的特性，同时，它还可以拥有自己的新特性。继承的目的都是为了代码重用。

继承的过程，就是从一般到特殊的过程。

（3）多态性。多态性是指不同类的对象对同一消息作出不同的响应。比如同样是加法，把两个复数加在一起和把两个整数加在一起肯定完全不同。又比如，同样是选择“编

辑"的"粘贴"操作,在文字处理程序和绘图程序中有不同的效果。

那么,多态的作用是什么呢? 可以这样理解,多态是为了实现接口重用。多态的作用,就是保证了基类和派生类的某一方法的方法名、参数和返回值格式完全相同,方便用户使用。

3.2 类的概念

类是对一些具有相同数据和操作的事物的描述。在 C# 中数据称为字段,操作称为方法。字段是对象的状态的抽象,方法是对象的行为的抽象。类的成员包含这两部分内容:

- 字段成员,它存储与类的实例相关的数据。
- 方法成员,它是执行代码。方法成员通常模拟类所表示的现实世界事物的功能和操作。

3.2.1 类的声明

在 C# 中,类使用 class 关键字来声明,其语法格式如下:

```
类修饰符  class  类名
{类体 }
```

其中,关键字 class、类名和类体是必需的,其他项是可选项。类修饰符是用于限定类申明的一种符号。按功能可分为两部分:访问修饰符和其他修饰符。

访问修饰符有四个,分别是:

- public:公共访问是允许的最高访问级别,对访问没有限制。
- private:私有访问是允许的最低访问级别,私有类只有在它们的类和结构体中才是可访问的。
- internal:内部类型,只有在同一文件中的类型才是可访问的。如果没有类修饰符,系统默认该选项。
- protected:受保护类,在其所声明的类中可由其所声明类的派生类实例访问。

其他修饰符包括 abstract、sealed、static 和 partial,其作用如下:

- abstract:声明虚类,指示某个类只能是其他类的基。
- sealed:指定类不能被继承。
- static:声明静态类,类型本身只含有静态成员,不能被实例化。
- partial:指示类、结构或接口可以写成几个部分。

这些类修饰符将在后续章节分别介绍。

类体用于定义类的成员,即字段成员和方法成员。

下面是一个简单的类声明和使用实例。大括号内包含了成员的声明,它们组成了类主体。类成员可以在类主体内部以任何顺序声明。这意味着成员的声明完全可以引用另一个在后面的类声明中才定义的成员。

例 3-1 设计一个 Circle(圆)类,其属性有圆心坐标 x 和 y、半径 r。其方法为 Set(int, int, double)和 void Print(),实现并测试这个类。

```
01: using System;
02:
03: namespace CSHARP3_1
04: {
05:     class Program
06:     {
07:         static void Main(string[] args)
08:         {
09:             Circle p= new Circle();
10:             p.SetPoint(30,50,10);
11:             Console.Write("Circle p: ");
12:             p.Print();
13:         }
14:     }
15:     class Circle
16:     {
17:         private double x,y,r;
18:         public void SetPoint(double a,double b,double c)
19:         {
20:             x= a;
21:             y= b;
22:             r= c;
23:         }
24:         public void Print()
25:         {
26:             Console.WriteLine("["+ x+ ","+ y+ "],"+ "Radius= "+ r);
27:         }
28:     }
29: }
```

运行结果:

```
Circle p: [30,50],Radius= 10
```

程序第 9 行通过关键字 new 创建了一个日期类对象,第 10 行代码通过 init 方法初始化了该对象的成员数据变量;第 17 行定义了三个私有数据成员 x(圆心坐标)、y(圆心坐标)和 r(圆半径),第 18～27 行定义了两个公有成员方法 SetPoint 和 Print。

这里定义了一个新的数据类型,为用户自己定义的数据类型,是对圆的特性和行为的描述,类型名为 Circle,和 double、char 等一样为一种数据类型。用定义新数据类型 Circle 类的方法把数据和处理数据的函数封装起来。

3.2.2　类成员的声明

C＃在类型的外部不能声明全局变量。所有的数据成员都属于某种类型，而且必须在类型声明内部声明。在类的内部包含字段成员和方法成员。访问修饰符是成员声明的可选部分，指明程序的其他部分如何访问成员。访问修饰符放在简单声明形式之前。下面是字段和方法声明的语法：

字段

```
成员修饰符 数据类型 标识符;
```

方法

```
成员修饰符 返回值类型 类型方法名 ()
{… }
```

类成员修饰符可分为访问修饰符和其他修饰符。访问修饰符有 5 种，如下所示：

- 私有的（private），系统缺省值是 private。
- 公有的（public）。
- 保护的（protected）。
- 内部的（internal）。
- 受保护内部的（protected internal）。

本章将阐述前两种 private（私有）和 public（公有）。

private 声明私有成员，私有字段成员只能被类内部的方法使用和修改，私有方法成员只能被类的内部方法调用。派生类可以继承基类的私有成员，但不能直接访问它们，只能通过基类的公有成员访问。

public 声明公有成员，类的公有方法成员和字段成员均可以被类的外部程序直接使用。公有方法实际是一个类和外部通信的接口，外部方法通过调用公有方法，按照预先设定好的方法修改类的私有成员和保护成员。

类成员的其他修饰符包括：

- abstract：指示该方法或属性没有实现需要在派生类中实现。
- const：指定域或局部变量的值不能被改动。
- event：声明一个事件。
- extern：指示方法在外部实现。
- override：对由基类继承成员的新实现。
- readonly：指示一个域只能在声明时以及相同类的内部被赋值。
- static：指示类的静态成员，为全体类的对象所共有。

例 3-2　定义盒子 Box 类。要求具有以下成员：可设置盒子形状 SetBox；可提供盒子体积 Volume；可提供盒子表面积 Area。源代码如下：

```
using System;

namespace CSHARP3_2
```

```
04: {
05:     class Program
06:     {
07:         static void Main(string[] args)
08:         {
09:             Box b=new Box();
10:             b.SetBox(10,10,5);
11:             b.Print();
12:         }
13:     }
14:     class Box
15:     {
16:         private double x=0,y=0,z=0;
17:         public void SetBox(double a,double b,double c)
18:         {
19:             x=a; y=b; z=c;
20:         }
21:         public double Area()
22:         {
23:             return 2* (x* y+y* z+x* z);
24:         }
25:         public double Volume()
26:         {
27:             return x* y* z;
28:         }
29:         public void Print()
30:         {
31:             Console.WriteLine("表面积: {0},体积: {1}",Area(),Volume());
32:         }
33:     }
34:
35: }
```

程序运行结果：

```
表面积: 400,体积: 500
```

程序 16 行通过用逗号分隔名称的方式，在同一条语句中声明多个相同类型的字段，但不能在一个声明中混合不同的类型。

3.2.3 类的字段

字段是类的数据成员，可以是 int、double、float、string 等类型。和所有变量一样，它可以被写入或读取。字段的初始化是字段声明的一部分，由一个等于号后面跟着一个求值表达式组成，初始化值在编译时决定。例 3-2 中第 14～16 行代码，决定了字段 x、y、z

的值。

```
    class Box
    {
        private double x=0,y=0,z=0;
        ⋮
    }
```

如果没有初始化语句,数据成员的值会被编译器设为默认值。默认值由字段的类型决定。简单总结起来就是每种类型的默认值都是 0,bool 型默认值是 false,引用类型默认值为 null。例如:

```
class Person
{
    string Name;            //初始化为 null,引用类型
    int Age;                //初始化为 0,值类型
    char Gender ;           //初始化为 0,值类型
}
```

3.2.4 创建类的实例

对象是类的实例,声明了类之后,就可以用类名定义该类的对象。一般来说,一个对象就是一个具有某种类型的变量。与普通变量一样,对象也必须经过声明才可以使用。声明对象的方法格式如下:

```
类名 对象名=new 类名();
```

例如:

```
Circle p=new Circle();
```

声明了一个名为 p 的 Circle 类的对象。创建对象的命令可以分解为两个步骤,第一步先声明引用变量,第二步为类对象分配内存。例如:

```
Circle p;                  //声明引用变量
p=new Circle();            //为类对象分配内存
```

数据的引用保存在一个类类型的变量中。要创建类的实例,需要从声明一个类类型的变量开始。如果变量没有被初始化,它的值是 null。声明类类型的变量所分配的内存是用来保存引用的,而不是用来保存类对象实际数据的。要为实际数据分配内存,需要使用 new 运算符。new 运算符为任意指定类型的实例分配并初始化内存。

例 3-3 声明了一个 Date 类,包含初始化 init 方法和输出 Printymd 和方法。

```
using System;

namespace CSHARP3_3
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Date date1=new Date();
            date1.init(2016,2,28);
            Console.Write("date1:");
            date1.Printymd();
        }
    }
    class Date
    {
        int day=1,month=1,year=1900;
        public void init(int yy,int mm,int dd)
        {
            month=(mm>=1 && mm<=12) ? mm : 1;
            year=(yy>=1900 && yy<=2100) ? yy : 1900;
            day=(dd>=1 && dd<=31) ? dd : 1;
        }
        public void Printymd()
        {
            Console.WriteLine(year+"-"+month+"-"+day);
        }
    }
}
```

程序的运行结果：

```
date1:2016-2-28
```

3.2.5 类的方法

类的方法是具有名称的可执行代码块，可从程序的很多不同地方执行，甚至可从其他程序中执行。当方法被调用时，它执行自己所含的代码，然后返回到调用它的代码。有些方法返回一个值到其被调用的位置。方法声明的具体格式为：

```
方法修饰符 返回值类型 方法名(形参列表)
{ 方法体 }
```

访问修饰符和字段的访问修饰符相同。返回类型声明了方法返回值的类型。如果一个方法不返回值，那么返回类型被指定为 void。方法名是方法的名称。参数列表由一对圆括号组成。如果有参数(参数将在下一章阐述)，它们被列在圆括号中间，也可以有空的参数列表。方法由一对大括号组成，大括号内包含执行代码。

例 3-4 定义一个 Person 类，数据成员有姓名 Name、年龄 Age、性别 Gender，成员方法有输出方法 SetPerson 和 PrintPerson，然后调用主程序验证 Person 类。

```
using System;
namespace CSHARP3_4
{
    class Program
    {
        static void Main(string[] args)
        {
            Person person1= new Person();
            person1.SetPerson("张三",19,'m');
            Console.Write("person1: ");
            person1.PrintPerson();
            Person person2= new Person();
            person2.SetPerson("李四",18,'f');
            Console.Write("person2: ");
            person2.PrintPerson();
        }
    }
    class Person
    {
        string Name= "XXX";
        int Age= 0;
        char Gender= 'm';
        public void SetPerson(string name,int age,char sex)
        {
            Name= name;
            Age= age;
            Gender= (sex == 'm' ? 'm' : 'f');
        }
        public void PrintPerson()
        {
            string str= (Gender == 'm' ? "男" : "女");
            Console.WriteLine("{0} \t {1} \t {2} ",Name,Age,str);
        }
    }
}
```

程序运行结果：

```
person1: 张三    19    男
person2: 李四    18    女
```

在程序中，可以用 person1.方法名或 person1.数据成员名访问对象的成员。例如：

```
person1.SetPerson("张三",19,'m')
```

3.2.6 类的构造方法

在建立类的对象时,需做一些初始化工作,例如对数据成员初始化。这些可以用构造方法来完成。构造方法是类的一个特殊的成员方法,它会在每次生成类对象时自动被调用。构造方法的说明格式如下:

```
修饰符 类名();
```

或者

```
修饰符 类名(参数表);
```

例如可以定义 Person 类没有参数的构造方法如下:

```
public Person(){                    //构造方法
    Name="张三";
    Age=19;
    Gender='m';
}
```

每当用 new 生成类的对象时,自动调用类的构造方法。因此,可以把初始化的工作放到构造方法中完成。构造方法和类名相同,没有返回值。当用

```
Person person1=new Person()
```

语句生成 Person 类对象时,将自动调用以上构造方法。

在 C# 语言中,同一个类中的方法,如果方法名相同,而参数的类型或个数不同,则认为是不同的方法,这叫方法重载。仅返回值不同,不能看作不同的方法。这样,可以在类定义中,定义多个构造方法,其名字相同,参数类型或个数不同。根据生成类的对象方法,调用不同的构造方法。

例如可以定义 Person 类的构造方法如下:

```
public Person(string name,int age,char sex){          //构造方法
    Name=name;
    Age=age;
    Gender=(sex=='m' ? 'm' : 'f');
}
```

调用语句为

```
Person person2=new Person("李四",18,'f')
```

用语句 Person person1=new Person( )生成对象时,调用无参数的构造方法;而用语句 Person person2=new Person("李四",18,'f')生成对象时,将调用有参数的构造方法。

例 3-5 定义一个带构造方法的 Person 类,数据成员有姓名 Name、年龄 Age、性别 Gender,成员方法有输出方法 PrintPerson,然后调用主程序验证 Person 类。

```
01: using System;
```

```
02: namespace CSHARP3_5
03: {
04:     class Program
05:     {
06:         static void Main(string[] args)
07:         {
08:             Person person1=new Person();
09:             Console.Write("person1: ");
10:             person1.PrintPerson();
11:             Person person2=new Person("李四",18,'f');
12:             Console.Write("person2: ");
13:             person2.PrintPerson();
14:         }
15:     }
16:     class Person
17:     {
18:         string Name="XXX";
19:         int Age=0;
20:         char Gender='m';
21:         public Person()                                 //构造方法
22:         {
23:             Name="张三";
24:             Age=19;
25:             Gender='m';
26:         }
27:         public Person(string name,int age,char sex)     //构造方法
28:         {
29:             Name=name;
30:             Age=age;
31:             Gender=(Gender=='m' ? 'm' : 'f');
32:         }
33:         public void PrintPerson()
34:         {
35:             string str=(Gender=='m' ? "男" : "女");
36:             Console.WriteLine("{0} \t {1} \t {2} ",Name,Age,str);
37:         }
38:     }
39: }
```

程序的运行结果：

```
person1: 张三    19    男
person2: 李四    18    女
```

代码第 18～20 行在编译时给类的字段赋初值“XXX”、0 和 m。

代码第 21～26 行在类的构造函数中对字段赋值“张三”、19 和 m。通过运行结果可以看出，类的构造函数是在程序编译后执行，因此结果输出如上所示。代码第 27～32 行也是同样道理。

3.3 类的属性

一般字段是类或结构中定义的数据成员。属性不是字段，本质上是定义修改字段的方法。C# 中的属性更充分地体现了对象的封装性：不直接操作类的数据内容，而是通过访问器进行访问，借助于 get 和 set 方法对属性的值进行读写。访问属性值的语法形式和访问一个变量基本一样，访问属性就像访问变量一样方便，符合编程习惯。

给属性赋值时使用 set 访问器，set 访问器始终使用 value 设置属性的值；获取属性值时使用访问器 get，get 访问器通过 return 返回属性的值。在访问声明中，如果只有 get 访问器，则表示是只读属性；如果只有 set 访问器，则表示是只写属性；如果既有 get 访问器，也有 set 访问器，则表示是读写属性。其格式为：

```
访问修饰符 数据类型 属性名
{
    get{
        get 访问器代码块
    }
    set{
        set 访问器代码块
    }
}
```

现在用属性来描述学生的总成绩。定义一个描述学生情况的类 Student，其中字段 ID 和 Namey 以及三门课的成绩是私有字段，记录学生的学号、姓名和三门课的成绩，通过属性给这五个私有字段赋值。

例 3-6 编写一个统计学生课程平均分的程序：输入两个学生的学号、姓名和三门课程的成绩（整型），统计每个学生三门课程的平均分，最后输出统计结果。代码如下：

```
using System;
namespace CSHARP3_6
{
    class Program
    {
        static void Main(string[] args)
        {
            Student student0=new Student();
            student0.Name="张三";
            student0.ID="2011";
            student0.ScoreMaths=80;
```

```
                student0.ScoreEnglish= 90;
                student0.ScorePE= 100;
                Student student1= new Student();
                student1.Name= "李四";
                student1.ID= "2012";
                student1.ScoreMaths= 70;
                student1.ScoreEnglish= 93;
                student1.ScorePE= 95;
                Console.WriteLine("---------------------------------------");
                Console.WriteLine("学号\t姓名\t高数\t英语\t体育\t平均分");
                Console.WriteLine("---------------------------------------");
                student0.PrintStudent();
                student1.PrintStudent();
            }
        }
        class Student
        {
            private string id= "0000";          //学号
            private string name= "xxx";         //姓名
            private int scoreMaths= 0;          //数学成绩
            private int scoreEnglish= 0;        //英语成绩
            private int scorePE= 0;             //体育成绩

            public string ID
            {
                get
                {
                    return id;
                }
                set
                {
                    id= value;
                }
            }

            public string Name
            {
                get
                {
                    return name;
                }
                set
                {
                    name= value;
```

```
            }
        }
        public int ScoreMaths
        {
            get
            {
                return scoreMaths;
            }
            set
            {
                scoreMaths= value;
            }
        }
        public int ScoreEnglish
        {
            get
            {
                return scoreEnglish;
            }
            set
            {
                scoreEnglish= value;
            }
        }
        public int ScorePE
        {
            get
            {
                return scorePE;
            }
            set
            {
                scorePE= value;
            }
        }

        public void PrintStudent()
        {
            Console.Write(ID+ "\t"+ Name+ "\t");
            Console.Write(ScoreMaths+ "\t"+ ScoreEnglish+ "\t"+ ScorePE+ "\t");
            Console.WriteLine((ScoreMaths+ ScoreEnglish+ ScorePE)/3);
        }
    }
}
```

运行结果：

```
----------------------------------------------------------
学号      姓名      高数      英语      体育      平均分
----------------------------------------------------------
2011      张三      80        90        100       90
2012      李四      70        93        95        86
```

3.4　自实现属性

很多时候，可以在 get 或 set 中加入代码。如在 set 中加入检查代码，只有符合要求时才能赋值。如果仅仅像上述例子一样，不需要检查，则可以使用 C＃的自动属性。不需要给出属性对应的 private 字段，由编辑器自动补上。需要注意的是自动属性必须既包含 get，也包含 set。用自动属性重写 class Student 的代码如下。

例 3-7　利用 C＃的自动属性，重新编写统计学生课程平均分的程序，观察输出结果。

```
using System;
namespace CSHARP3_7
{
    class Program
    {
        static void Main(string[] args)
        {
            Student student0= new Student();
            student0.Name= "张三";
            student0.ID= "2011";
            student0.ScoreMaths= 80;
            student0.ScoreEnglish= 90;
            student0.ScorePE= 100;
            Student student1= new Student();
            student1.Name= "李四";
            student1.ID= "2012";
            student1.ScoreMaths= 70;
            student1.ScoreEnglish= 93;
            student1.ScorePE= 95;
            Console.WriteLine("-----------------------------------------");
            Console.WriteLine("学号\t姓名\t高数\t英语\t体育\t平均分");
            Console.WriteLine("-----------------------------------------");
            student0.PrintStudent();
            student1.PrintStudent();
        }
    }
    class Student
    {
```

```
            public string ID { get; set; }
            public string Name { get; set; }
            public int ScoreMaths { get; set; }
            public int ScoreEnglish { get; set; }
            public int ScorePE { get; set; }
            public void PrintStudent()
            {
                Console.Write(ID+ "\t"+ Name+ "\t");
                Console.Write(ScoreMaths+ "\t"+ ScoreEnglish+ "\t"+ ScorePE+ "\t");
                Console.WriteLine((ScoreMaths+ ScoreEnglish+ ScorePE)/3);
            }

        }
}
```

在 Visual Studio 集成开发环境中可以在源代码中插入预定义代码模板。右击源代码编辑器并从菜单中选择"插入代码段"→visual C♯→prop 菜单项，或者使用快捷键 Ctrl+K、Ctrl+X。

3.5 值类型和引用类型

C♯语言的类型被分为两类——值类型和引用类型。这两种类型的对象在内存中的存储方式不同。数据类型不仅定义了存储数据需要的内存大小、组成该类型的数据成员以及该类型能执行的函数，还决定了对象在内存中的存储位置——栈或堆。

栈是一个内存数组，是一个 LIFO(last-in first-out，后进先出)的数据结构。栈存储几种类型的数据：

- 变量的值；
- 程序当前的执行环境；
- 传递给方法的参数。

堆是一块内存区域，在堆里可以分配大块的内存用于存储某类型的数据。与栈不同，堆里的内存可以任意顺序存入和删除。虽然程序可以在堆里保存数据，但并不能显式地删除它们。CLR 的自动 GC(Garbage Collector，垃圾收集器)在判断出程序的代码将不会再访问某数据项时，会自动清除无主的堆对象。

值类型只需要一段单独的内存，用于存储实际的数据。

引用类型需要两段内存：第一段存储实际的数据，它总是位于堆中。第二段是一个引用，指向数据在堆中的存放位置。

例如，假设有一个引用类型 Person 的实例，名称为 person1。它有两个成员：一个值类型成员 Age 和一个引用类型成员 Name。它将如何存储呢？是否值类型的成员 Age 存储在栈里，而引用类型的成员 Name 在栈和堆之间分成两半呢？答案是否定的。

对于一个引用类型，其实例的数据部分始终存放在堆里。既然两个成员都是对象数

据的一部分，那么它们都会被存放在堆里，无论它们是值类型还是引用类型。

如图 3-1 所示，尽管成员 Age 是值类型，但它也是 Person 实例数据的一部分，因此和对象的数据一起被存放在堆里。成员 Name 是引用类型，所以它的数据部分会始终存放在堆里。

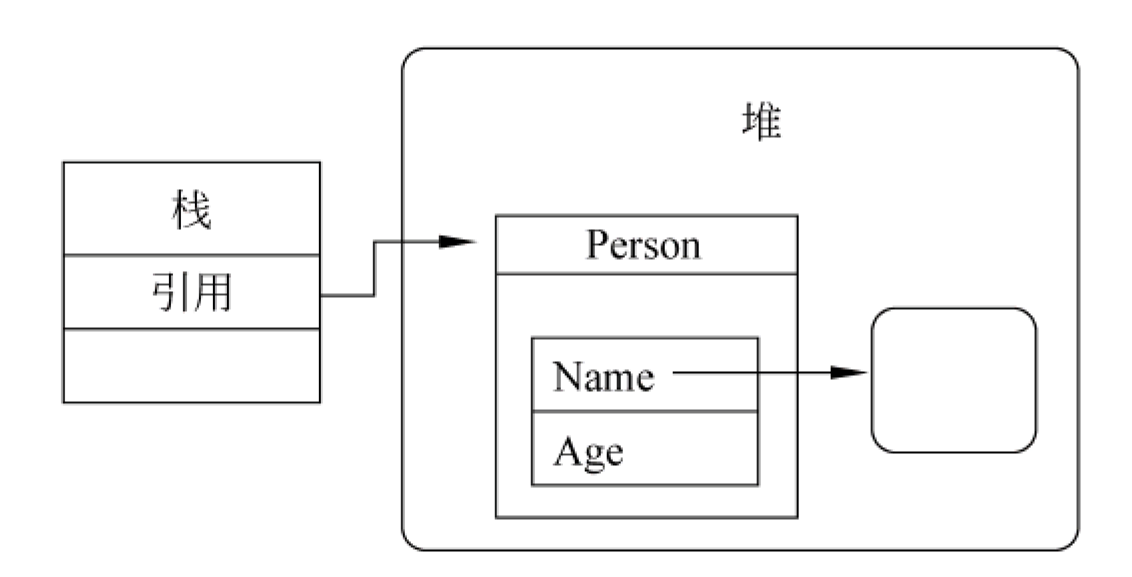


图 3-1　引用类型成员的数据存储

3.6　静态字段和实例字段

用修饰符 static 声明的字段为静态字段。不管包含该静态字段的类生成多少个对象或根本无对象，该字段都只有一个实例，静态字段不能被撤销。必须采用如下方法引用静态字段：

类名.静态字段名

如果类中定义的字段不使用修饰符 static，则该字段为实例字段。每创建该类的一个对象，在对象内就会创建一个该字段实例；创建它的对象被撤销，该字段对象也被撤销。实例字段采用如下方法引用：

实例名.实例字段名

例 3-8　定义 My 类，包括一个静态数据成员和一个静态成员方法；然后在主程序中生成两个 My 类的对象 x 和 y，本别调用 5 次 x 和 y 的静态成员方法，观察输出结果。

```
using System;

namespace CSHARP3_8
{
    class Program
    {
        static void Main(string[] args)
        {
            My x=new My();
            My y=new My();
            for (int i=0; i<5; i++)                //分别调用 5次 func()方法
            {
```

```
                Console.Write(My.func()+"\t");      //不能使用 x.func()
                Console.Write(My.func()+"\t");      //不能使用 y.func()
            }
            Console.WriteLine();
        }
    }
    class My
    {
        static int count=0;
        static public int func()
        {
            return++count;
        }
    }
}
```

程序输出：

```
1	2	3	4	5	6	7	8	9	10
```

例 3-9 将静态成员换成实例成员，观察程序运行结果。

```
using System;

namespace CSHARP3_7
{
    class Program
    {
        static void Main(string[] args)
        {
            My x=new My ();
            My y=new My();
            for (int i=0; i<5; i++)      //分别调用 5次 func() 方法
            {
                Console.Write(x.func()+"\t");
                Console.Write(y.func()+"\t");
            }
            Console.WriteLine();
        }
    }
    class My
    {
        int count=0;
        public int func()
        {
            return++count;
```

```
                }
        }
}
```

程序输出：

```
1	1	2	2	3	3	4	4	5	5
```

习题

1. 定义一个 Circle(椭圆)的类，其数据成员是椭圆外切矩形的左上角和右下角两点的坐标，定义计算面积和周长的成员方法，然后调用主程序验证。

2. 定义一个带构造方法的 Box(盒子)类。要求具有以下成员：可设置盒子形状，可提供盒子体积，可提供盒子表面积。

3. 定义并实现 Dog 类，其数据成员包含姓名 Name、年龄 Age、性别 Gender、体重 Weight，其方法包括初始化方法 Init 和显示方法 PrintDog，调用主程序验证 Dog 类。

4. 定义并实现地址类 Address，包括姓名、所居住的街道地址、城市和邮编等属性以及设置对象字段的 SetAddress 方法、显示地址信息的 Display 方法。

5. 定义并实现三维空间的 Point3D 类，包括 x、y、z 三个成员变量，一个计算空间中两个点之间距离的成员方法，请编写合适的构造方法。

6. 定义并实现一个公民类 Citizen，该类包括的特征信息有：身份证号 Id、姓名 Name、性别 Gender、年龄 Age、籍贯 Birthplace、家庭住址 Familyaddress 等属性以及构造方法、输入公民信息方法 Input 以及输出公民信息方法 Print，要求能够对该类对象进行初始化、输入和输出操作。

7. 定义并实现三角形类 Triangle，其成员变量包括三个边长变量，成员方法包括判断是否合法 isLegal、计算面积 Area，以及是否构成直角三角形、锐角三角形的钝角三角形 isType 等方法，然后调用主程序验证。

第4章 控制语句

语句是描述一个类型或告诉程序去执行一个动作的源代码指令。程序中的语句通常按编写的顺序一条一条地执行，称为顺序执行。程序也可以执行不是紧邻其后的语句，这称为控制执行。

4.1 程序的基本控制结构

C#是一种支持面向对象程序设计思想的程序设计语言。使用C#编写程序时，应该也遵循面向对象程序设计方法。按照这种原则和方法设计出的程序具有结构清晰、可读性好、易于修改和容易验证等优点。

按照程序设计的观点，任何算法功能都可以通过由程序模块组成的三种基本程序结构：顺序结构、选择结构和循环结构的组合来实现。

程序模块如图 4-1 所示。顺序结构由两个程序模块串接构成，如图 4-2 所示。由图 4-2 可以看出，这两个程序模块是顺序执行的，即首先执行“程序模块 1”，然后执行“程序模块 2”。

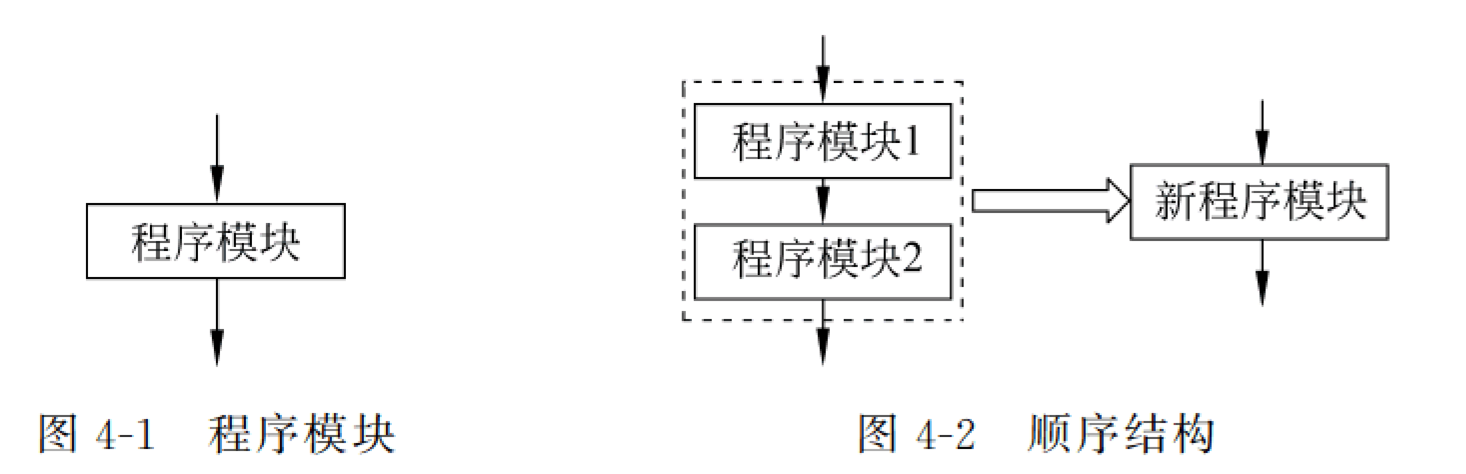


图 4-1 程序模块　　图 4-2 顺序结构

选择结构如图 4-3 所示。从图 4-3 可以看出，根据条件成立与否，分别选择执行“程序模块 1”或执行“程序模块 2”。虽然选择结构比顺序结构稍微复杂了一点，但是仍然可以将其整个作为一个新的程序模块：一个入口（从顶部进入模块开始判断）、一个出口（无论执行“程序模块 1”还是“程序模块 2”，都应从选择结构框的底部出去）。

在编程中，还可能遇到选择结构中的一个选择没有实际操作的情况，如图 4-4 所示。这种形式的选择结构可以看成是图 4-3 中的选择结构的特例。

循环结构如图 4-5 所示。在进入循环结构后首先判断条件是否成立，如果成立则执行“程序模块”，反之则退出循环结构。执行完“程序模块”后再去判断条件，如果条件仍然

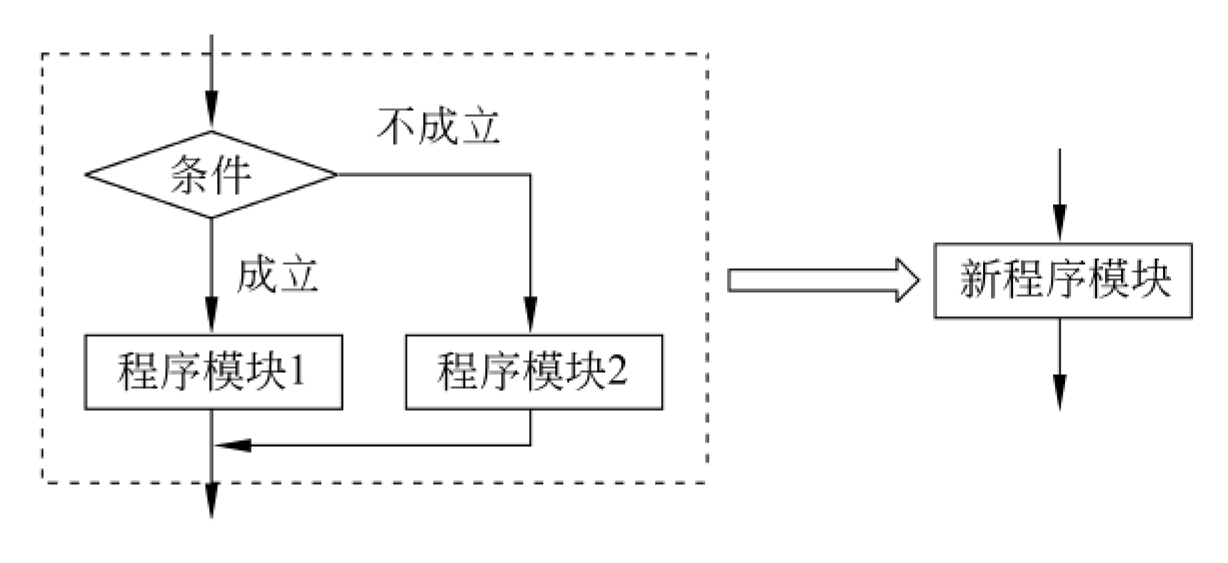


图 4-3　选择结构

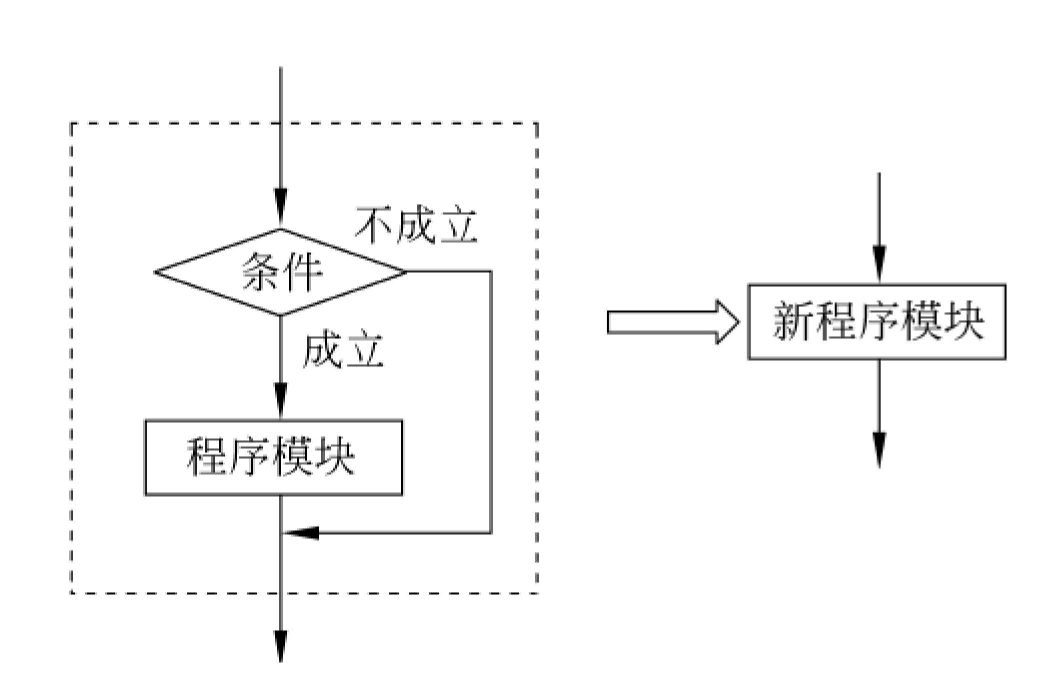


图 4-4　有一个分支没有实际操作的选择结构

成立,则再次执行"程序模块"。循环往复,直至条件不成立时退出循环结构。

与顺序和选择结构相同,循环结构也可以抽象为一个新的模块。图 4-5 中的循环结构可以描述为"当条件成立时反复执行程序模块",故又称为当型循环。除了当型循环以外,还有一种直到型循环结构,其特点是进入循环结构后首先执行"程序模块",然后再判断条件是否成立。如果成立,则再次执行"程序模块";如此反复,直到条件不成立时退出循环结构。直到型循环结构如图 4-6 所示。

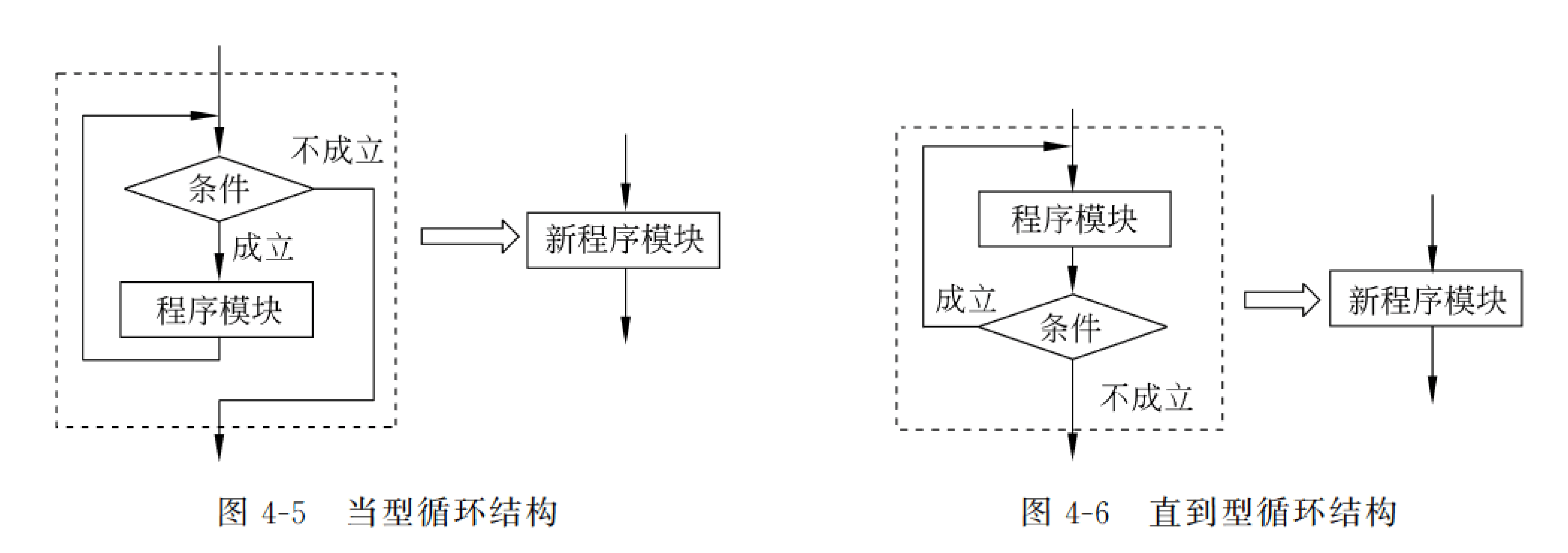


图 4-5　当型循环结构　　　　图 4-6　直到型循环结构

直到型循环和当型循环最大的不同是：直到型循环的循环体最少执行一次,而当型循环的循环体可能一次也不执行。这一点可以很容易地从图 4-5 和图 4-6 的比较中看出。

循环结构是程序中使用最多的一种结构,几乎所有的实用程序中都包含循环。它可以充分发挥计算机长于进行快速重复运算的特点。

4.2 控制语句

C#有三种基本控制语句：顺序语句、选择语句（三种）和循环语句（四种），跳转语句可以看作是选择语句或者是循环语句的一部分。每个C#程序都是根据程序所需要的算法组合这三种控制语句。

选择执行依据一个条件执行或跳过一个代码片段。选择执行语句如下：

- if
- if_else
- switch

循环语句重复执行一个代码片段。循环语句如下：

- while
- do
- for
- foreach

跳转语句把控制流从一个代码片段改变到另一个代码片段中的指定语句。跳转语句如下：

- break
- continue
- return
- goto
- throw

条件执行和循环结构（除了foreach）需要一个测试表达式或条件以决定程序应当在哪里继续执行。

4.3 选择语句

C#有三种选择语句，即if语句、if_else语句和switch语句。

4.3.1 if语句

if语句实现按条件执行。if语句的语法如下所示：

```
if(<表达式>)
    <语句>;
```

<表达式>的返回值必须是bool型。如果<表达式>求值为true，<语句>被执行。如果<表达式>求值为false，<语句>被跳过。

下列代码展示了if语句的示例：

```
if(x<=60)
```

```
    x=x+1;              //简单语句不需要大括号
if(x<80)                //使用分语句
{
    x=x-5;
}
int x=72;
if(x)                   //错,表达式应该是 bool 型,而不是 int 型
{
    ⋮
}
```

4.3.2　if…else 语句

C＃的选择结构是通过 if-else 语句实现的。其格式为：

```
if(<表达式>)
    <语句 1>;
else
    <语句 2>;
```

一般来说，语句 1 和语句 2 可以是各种语句，甚至包括 if-else 语句和后面要介绍的循环语句。如果“程序模块 1”和“程序模块 2”比较复杂，不能简单地用一条语句实现，则可以使用由一对花括号“{}”括起来的程序段落代替“语句 1”和“语句 2”，即：

```
if(<表达式>)
{
    ⋮
}
else
{
    ⋮
}
```

这种用花括号括起来的程序段落又称为分程序。分程序是 C＃中的一个重要概念。具体说来，一个分程序具有下述形式：

```
{
    <局部数据说明部分>
    <执行语句段>
}
```

即分程序是由花括号括起来的一组语句。当然，分程序中也可以再嵌套新的分程序。分程序是 C＃程序的基本单位之一。

分程序在语法上是一个整体，相当于一个语句。因此分程序可以直接和各种控制语句结合使用，用以构成 C＃程序的各种复杂的控制结构。在分程序中定义的变量的作用范围仅限于该分程序内部。

例 4-1 编程实现分段函数：

$$y=\begin{cases}x+1 & (x<0)\\ 1 & (0\leqslant x<1)\\ x^3 & (1\leqslant x)\end{cases}$$

程序代码如下：

```
using System;

namespace CSHARP4_1
{
    class Program
    {
        static void Main(string[] args)
        {
            double x,y;
            Console.WriteLine("请输入 x的值 : ");
            x=Convert.ToDouble(Console.ReadLine());
            if (x<0)
            {
                y=x+1;
                Console.WriteLine("x={0}, y=x+1={1}",x,y);
            }
            else if (x<1)           //0≤x<1
            {
                y=1;
                Console.WriteLine("x={0}, y={1}",x,y);
            }
            else                    //1≤x
            {
                y=x*x*x;
                Console.WriteLine("x={0}, y=x*x*x={1}",x,y);
            }
        }
    }
}
```

运行结果：

```
请输入 x的值 :
15
x=15, y=x*x*x=3375
```

4.3.3 switch 语句

switch 语句用于实现多重分支，其格式为：

```
switch (<整型表达式>)
{
    case <数值 1>:
        ...; break;
    case <数值 2>:
        ...; break;
    case <数值 3>:
        ...; break;
    default:
        ...; break;
}
```

其中 default 模块也可省略。

switch 语句的执行过程是：首先计算整型表达式的值，然后将其结果与每一个 case 后面的数值常量依次进行比较；如果相等，则执行该 case 模块中的语句，否则依次执行其后每一个 case 模块中的语句。在 case 模块的最后必须加上一个 break、return 或 throw 语句，这样才能实现真正的多路选择。如果整型表达式的值与所有 case 模块的进入值无一相同，则执行 default 模块中的语句。在 case 语句的结尾忘记 break 语句时常产生逻辑错误。图 4-11 为带有 break 语句的 switch 多选择结构的框图。

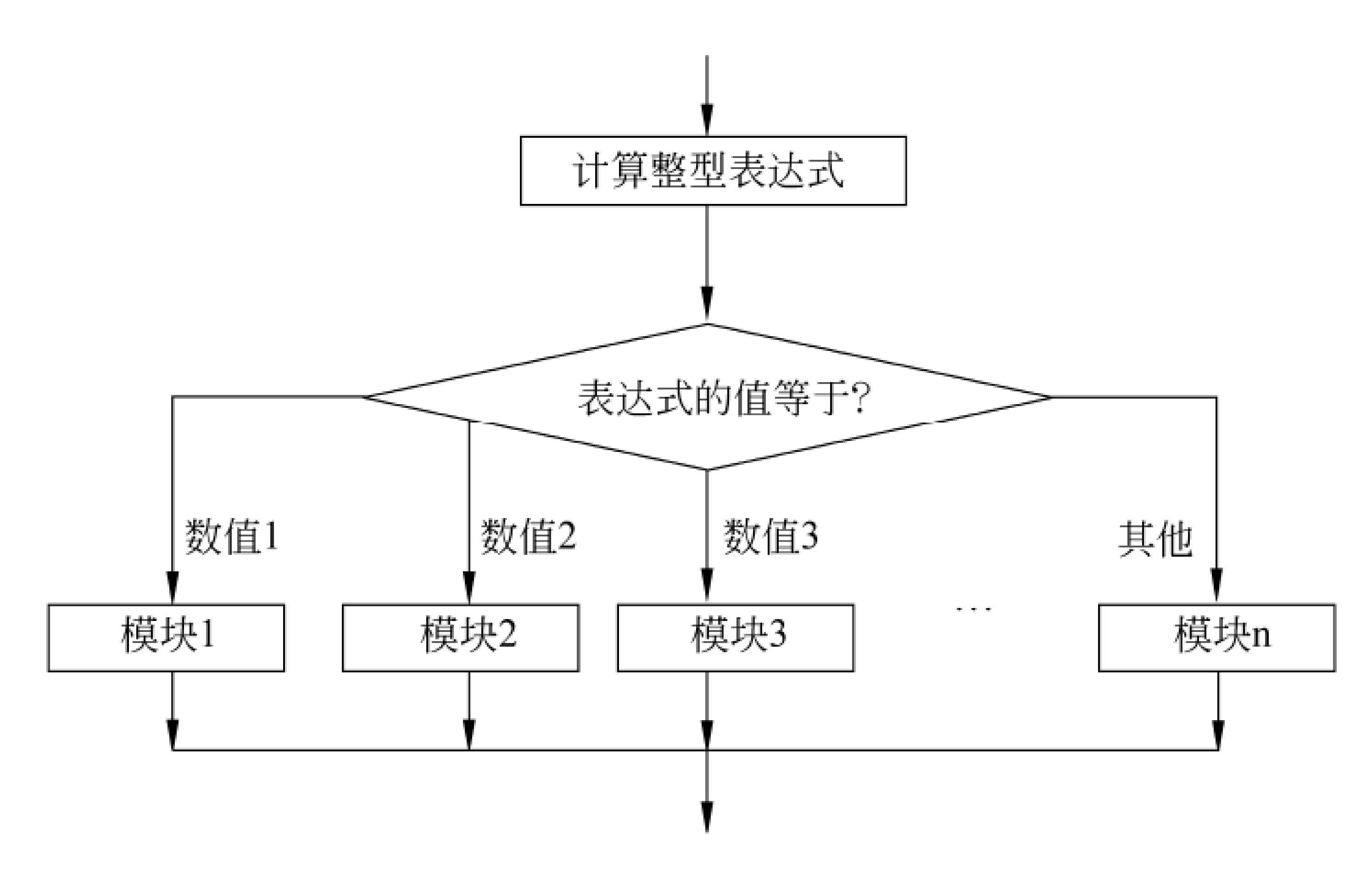


图 4-7 switch 多路选择结构

例 4-2 编写一个程序，将百分制的学生成绩转换为优秀、良好、中等、及格和不及格的 5 级制成绩。标准如下：

优秀：100～90 分；

良好：80～89 分；

中等：70～79 分；

及格：60～69 分；

不及格：60 分以下。

使用 switch 语句构成的多选择结构编写这个程序。switch 语句根据具体的数值判

断执行的路线，而现在的转换标准是根据分数范围。因此，构造一个整型表达式 old_grade/10 用于将分数段化为单个整数值。例如对于分数段 60～69 中的各分数值，上述表达式的值均为 6。再配合以在 switch 语句的各 case 模块中灵活运用 break 语句，即可编写出所需转换程序。

```
using System;

namespace CSHARP4_2
{
    class Program
    {
        static void Main(string[] args)
        {
            int old_grade;
            string new_grade;
            Console.Write("请输入学生成绩：");
            old_grade=Convert.ToInt32(Console.ReadLine());
            switch (old_grade/10)
            {
                case 10:
                case 9:
                    new_grade="优秀"; break;
                case 8:
                    new_grade="良好"; break;
                case 7:
                    new_grade="中等"; break;
                case 6:
                    new_grade="及格"; break;
                default:
                    new_grade="不及格"; break;
            }
            Console.WriteLine("转换前成绩是{0},
                     转换后成绩是{1}",old_grade,new_grade);
        }
    }
}
```

程序运行结果：

```
请输入学生成绩：85
转换前成绩是 85,转换后成绩是良好
```

该程序将用户输入的百分制的分数值转换为 5 级制成绩。请注意 switch 语句的第 1 个 case 模块中没有任何语句（包括 break），因此进入该模块时（原成绩为 100 分）将直接转入第 2 个 case 模块（处理原成绩在 90～99 分之间情况）中继续执行。

4.4　循环语句

4.4.1　while 语句

当型循环结构可以使用 while 语句实现：

```
while (<表达式>)
    <循环体>
```

其中的＜循环体＞可以是一个语句，也可以是一个分程序：

```
while(<表达式>)
{
    ⋮
}
```

while 语句的执行过程见图 4-5。当表达式的结果不为 0 时反复执行其循环体内的语句或者分程序，直到表达式的值为 0 时退出循环。所以在设计当型循环时要注意在其循环体内应该有修改＜表达式＞的部分，以此确保在执行了一定次数之后可以退出循环，否则循环永不结束，就成了“死循环”。

例 4-3　计算 $e=1+\frac{1}{1!}+\frac{1}{2!}+\cdots++\frac{1}{n!}+\cdots$，当通项$\frac{1}{n!}<10^{-7}$时停止计算。

定义三个工作变量 e、n 和 u，分别用于存放已计算出的结果近似值、当前项序号和当前通项值，则伪代码算法为：

```
e=1.0; n=1; u=1.0;
while(通项 u大于等于 10^-7)
{
    计算新的通项值 u=u/n;
    将新通项值加到结果近似值上;
        准备处理下一项 n=n+1;
}
```

程序代码如下：

```
using System;

namespace CSHARP4_3
{
    class Program
    {
        static void Main(string[] args)
        {
            double e=1.0;
            double u=1.0;
```

```
                int n=1;
                while (u>=1.0E-7)
                {
                    u=u/n;
                    e=e+u;
                    n=n+1;
                }
                Console.WriteLine("e={0} (n={1})",e,n);
            }
        }
}
```

程序运行结果：

```
e=2.71828182619849 (n=12)
```

根据计算结果同时打印出项数 n，表明该级数收敛相当快，仅计算到前 12 项其截断误差便已小于 10^{-7}。

4.4.2 do…while 语句

直到型循环结构可以使用 do…while 语句实现：

```
do
{
    <循环体>
}while (<表达式>);
```

例 4-4 使用 do…while 结构重新编写例 4-3 的程序。

程序代码如下：

```
using System;

namespace CSHARP4_4
{
        class Program
        {
            static void Main(string[] args)
            {
                double e=1.0;
                double u=1.0;
                int n=1;
                do
                {
                    u=u/n;
                    e=e+u;
                    n=n+1;
```

```
            } while (u >= 1.0E-7);
            Console.WriteLine("e= {0} (n= {1})",e,n);
        }
    }
}
```

4.4.3 for 语句

C#还提供了一种使用起来更为方便灵活的 for 语句,其控制流程如图 4-8 所示。格式为:

```
for (<表达式 1>; <表达式 2>; <表达式 3>)
    <循环体>
```

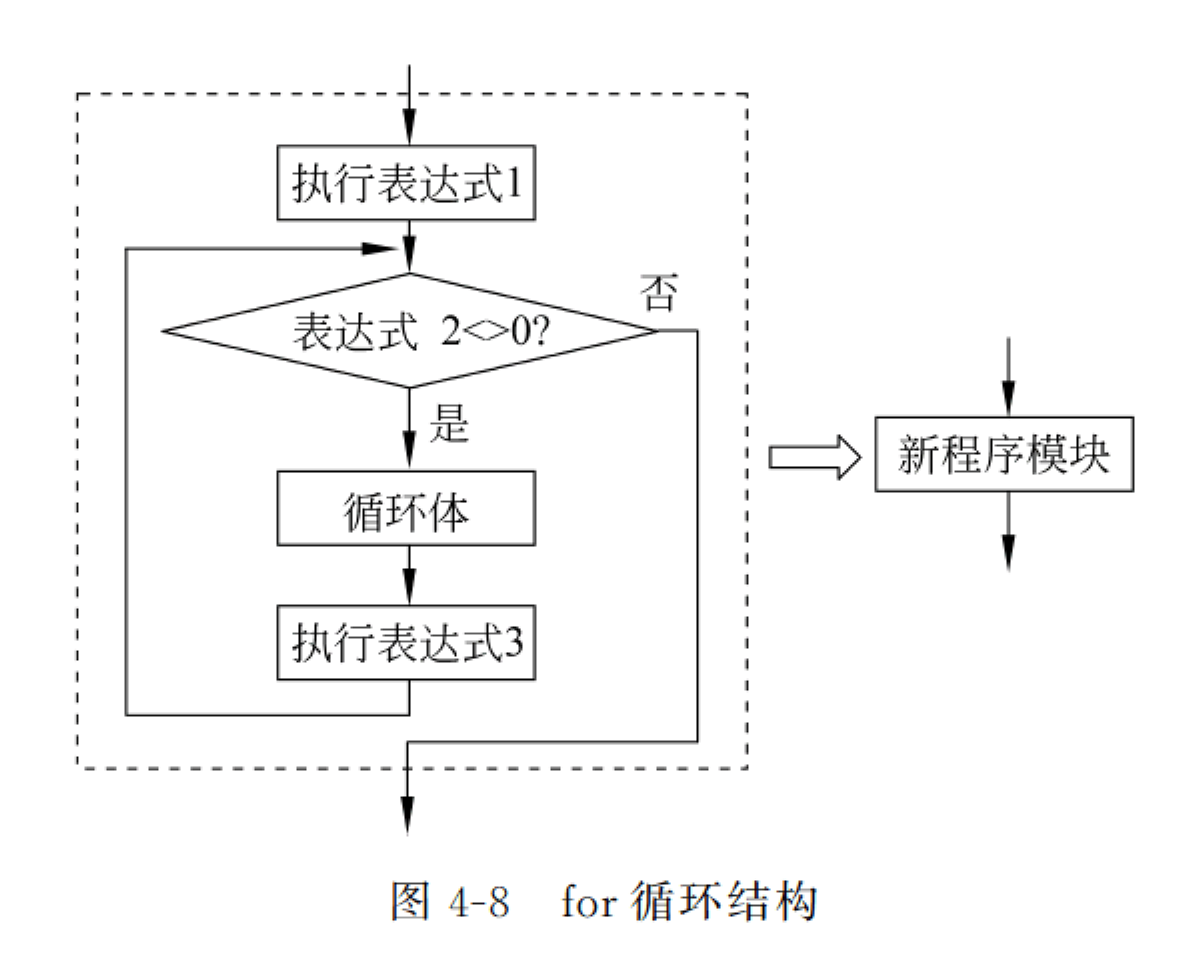


图 4-8 for 循环结构

和 while 语句的情况类似,for 语句的循环体也可以是一条语句或者一个分程序。for 语句最常见的用途是构造指定重复次数的循环结构。例如:

```
for (i= 0; i< 10; i= i+ 1)
{
    …
}
```

用于实现重复 10 次的循环。虽然用 while 语句和 do-while 语句也可以构造出这样的循环,但使用 for 语句更简单、直观。特别是在处理数组时,大多数程序员都喜欢使用 for 语句。

例 4-5 求水仙花数。如果一个三位数的个位数、十位数和百位数的立方和等于该数自身,则称该数为水仙花数。编写程序求出所有的水仙花数。在程序中利用了 C# 的整数除法和求余运算从一个 3 位数中分离出其个位、十位和百位数。

程序代码如下:

```
using System;

```

```
namespace CSHARP4_5
{
    class Program
    {
        static void Main(string[] args)
        {
            int n,i,j,k;
            for (n=100; n<=999; n++)
            {
                i=n/100;                          //取出 n的百位数
                j=(n/10)%10;                      //取数 n的十位数
                k=n%10;                           //取出 n的个位数
                if (n==i*i*i+j*j*j+k*k*k)
                    Console.WriteLine("{0}={1}^3+{2}^3+{3}^3",n,i,j,k);
            }
        }
    }
}
```

程序运行结果：

```
153=1^3+5^3+3^3
370=3^3+7^3+0^3
371=3^3+7^3+1^3
407=4^3+0^3+7^3
```

4.4.4 循环的嵌套

当循环语句中的循环体又包含另一个循环语句时，就构成了嵌套循环。循环的嵌套层次从语法上没有限制，但一般不超过 3 层，否则将影响可读性。

例 4-6 编写程序制作九九乘法表。

程序代码如下：

```
using System;

namespace CSHARP4_6
{
    class Program
    {
        static void Main(string[] args)
        {
            int i,j;
            for (i=1; i<10; i++)
            {
                for (j=1; j<=i; j++)
```

```
                        Console.Write("{0}* {1}={2}\t",j,i,i* j);
                    Console.WriteLine();
                }
            }
        }
}
```

程序运行结果：

```
1* 1=1
1* 2=2  2* 2=4
1* 3=3  2* 3=6   3* 3=9
1* 4=4  2* 4=8   3* 4=12  4* 4=16
1* 5=5  2* 5=10  3* 5=15  4* 5=20  5* 5=25
1* 6=6  2* 6=12  3* 6=18  4* 6=24  5* 6=30  6* 6=36
1* 7=7  2* 7=14  3* 7=21  4* 7=28  5* 7=35  6* 7=42  7* 7=49
1* 8=8  2* 8=16  3* 8=24  4* 8=32  5* 8=40  6* 8=48  7* 8=56  8* 8=64
1* 9=9  2* 9=18  3* 9=27  4* 9=36  5* 9=45  6* 9=54  7* 9=63  8* 9=72  9* 9=81
```

计算机首先按行输出，从第 1 行到第 9 行，输出数据中第 3 列正好是 1～9。每行输出后中的数据是由第 2 层循环实现的，其中第 1 个数就是列数，比如在第 5 行，请寻找数字 1、2、3、4、5，正好是列数。

4.5 跳转语句

C＃提供的控制转移语句，除了前面介绍的 if-else 语句、while 语句、do-while 语句和 for 语句以外，还有如下一些控制语句。

4.5.1 break 语句

break 语句的格式为：

```
break;
```

将该语句用在 switch 语句中，可以使程序流程跳出 switch 结构。如果将它用于循环语句，可以使流程立即跳出包含该 break 语句的各种循环语句，即提前结束循环，接着执行循环下面的语句。在循环语句中使用 break 语句时，一般应和 if 语句配合使用。例如：

```
while(<条件 1>)
{
    …
    if(<条件 2>)
        break;
    …
}
```

以上结构的框图如图 4-9 所示。

例 4-7 给定一个正整数,判定其是否为素数。

程序代码如下:

```
using System;

namespace CSHARP4_7
{
    class Program
    {
        static void Main(string[] args)
        {
            int x,i,m;
            Console.Write("请输入整数:");
            x=Convert.ToInt32(Console.ReadLine());
            m=Convert.ToInt32(Math.Sqrt(x));
            for (i=2; i<=m; i++)
            {
                if (x %i ==0)
                    break;
            }
            if (i >m)
                Console.WriteLine("{0}是素数",x);
            else
                Console.WriteLine("{0}不是素数",x);
        }
    }
}
```

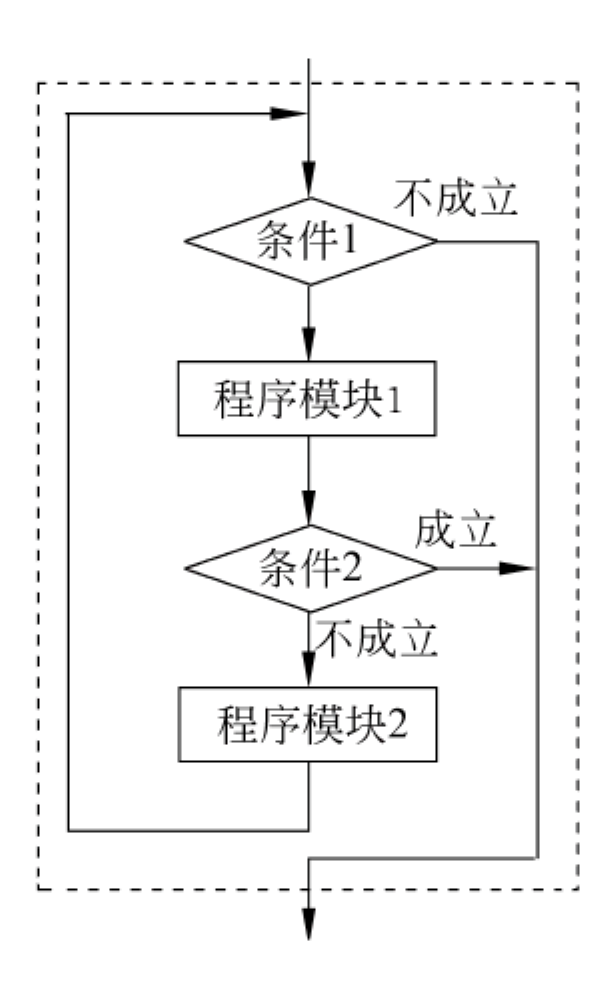


图 4-9 使用 break 语句的循环结构

运行结果:

```
请输入整数:2
2是素数
```

4.5.2 continue 语句

continue 语句用于提前结束本轮循环,即跳过循环体中下面尚未执行的语句,接着进行下一次是否执行循环的判断,可用于 while、do-while 和 for 语句中。其格式为:

```
continue;
```

continue 语句的用法和 break 语句相似,均应和 if 语句配合使用。仍以 while 语句为例:

```
while(<条件 1>)
{
```

```
    ⋮
    if(<条件 2>)
        continue;
    ⋮
}
```

其执行框图如图 4-10 所示。

在循环中使用 break 语句和 continue 语句的区别是：break 语句是结束整个循环的执行，再不进行条件判断；而 continue 语句则只结束本次循环，而不终止整个循环过程。

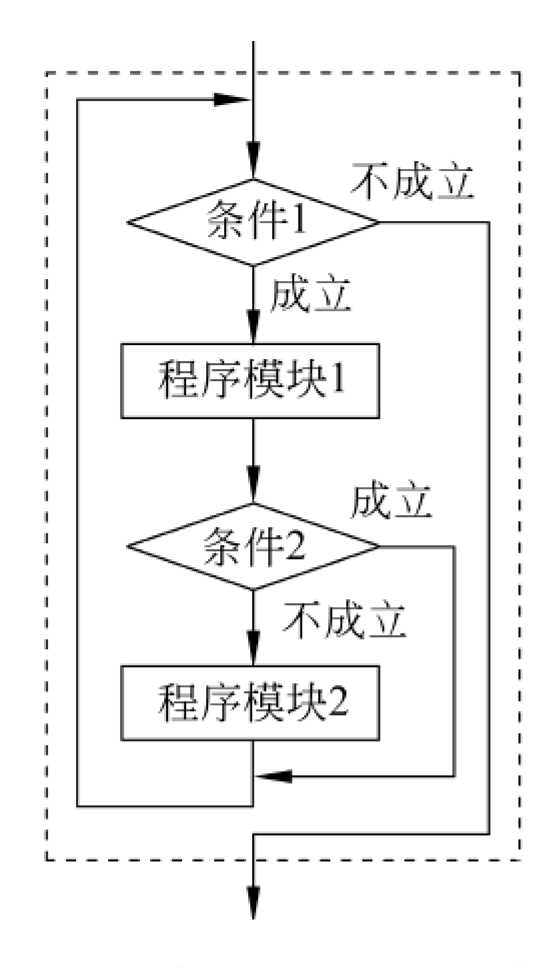


图 4-10 使用 continue 语句的循环结构

例 4-8 输入一批考试成绩，用－1 作为结束标志。如输入的成绩大于 100，则提示重新输入。然后计算平均分。

程序代码如下：

```
using System;

namespace CSHARP4_8
{
    class Program
    {
        static void Main(string[] args)
        {
            int x,count=0;
            double sum=0;
            Console.Write("请输入成绩：");
            do
            {
                x=Convert.ToInt32(Console.ReadLine());
                if (x > 100)
                {
                    Console.Write("成绩大于 100,请重新输入：");
                    continue;
                }
                else if (x == -1)
                    break;
                sum=sum+x;
                count=count+1;
            } while (true);
            Console.WriteLine("平均成绩={0}",sum/count);
        }
    }
}
```

运行结果：

```
请输入成绩：60
70
80
123
成绩大于 100,请重新输入：90
-1
平均成绩=75
```

4.5.3 goto 语句和语句标号

C#允许在语句前面放置一个标号，其一般格式为：

```
<标号>：<语句>；
```

标号的取名规则和变量名相同，即由下划线、字母和数字组成，第一个字符必须是字母或下划线。例如：

```
ExitLoop: x=x+1;
End: return x;
```

在语句前面加上标号主要是为了使用 goto 语句。goto 语句的格式为：

```
goto <标号>;
```

其功能是改变语句执行顺序，转去执行前面有指定标号的语句，而不管其是否排在当前语句之后。C#的 goto 语句只能在本函数模块内部进行转移，不能由一个函数中转移到另一个函数中去。

例 4-9 求 π 的近似值。利用公式：$\frac{\pi}{4}\approx 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$计算 π 的近似值，直到最后一项的绝对值小于指定数值(1.0e−7)为止。

程序代码如下：

```
using System;

namespace CSHARP4_9
{
    class Program
    {
        static void Main(string[] args)
        {
            int s=1;
            double n=1.0,u=1.0,pi=0.0;
        label1:
            if (Math.Abs(u) <=1.0e-7)
                goto label2;
            else
```

```
                {
                    u=s/n;
                    pi=pi+u;
                    n=n+2;
                    s=-s;
                    goto label1;
                }
            label2:
                Console.WriteLine("pi={0}",4*pi);
            }
        }
}
```

运行结果：

```
pi=3.14159
```

4.6 其他语句

其他控制语句和语言的特殊特征相关，这些语句在涉及那些特征的章节中阐述。主要语句包括：

- checked，unchecked：控制溢出检查上下文。
- foreach：遍历一个集合的每个成员。
- try，throw，finally：和异常有关。
- return：返回控制到调用函数成员，而且还能返回一个值。
- using：减少意外运行时错误带来的潜在问题。
- yield：用于迭代。

4.7 程序设计实例

例 4-10 找出 2～10000 之内的所有完全数。所谓完全数是该数各个真因子之和等于它本身的自然数，又称完美数或完备数(perfect number)。例如：第一个完全数是 6，它有约数 1、2、3、6，除去它本身 6 外，其余 3 个数相加：1＋2＋3＝6。第二个完全数是 28，它有约数 1、2、4、7、14、28，除去它本身 28 外，其余 5 个数相加：1＋2＋4＋7＋14＝28。后面的完全数还有 496、8128 等。

程序首先计算整数的真因子之和。整数 1 肯定是因子，不需要判断，直接赋给因子之和 sum。计算整数 i 的真因子之和算法如下。

```
sum=1;
for(j=2;j<=i/2;j++)
{
    如果 j 是 i 的因子,将 j 加入因子之和 sum 中
```

```
}
```

然后根据整数 i 是否等于真因子之和 sum 来判断 i 是否是完全数。

程序代码如下：

```
using System;

namespace CSHARP4_10
{
    class Program
    {
        static void Main(string[] args)
        {
            int i,j,sum;
            for (i=2; i<10000; i++)
            {
                sum=1;
                for (j=2; j<=i/2; j++)
                {
                    if (i%j==0)
                        sum=sum+j;
                }
                if (sum==i)
                {
                    Console.Write("{0}=1",i);
                    for (j=2; j<=i/2; j++)
                    {
                        if (i%j==0)
                            Console.Write("+{0}",j);
                    }
                    Console.WriteLine();
                }
            }
        }
    }
}
```

运行结果：

```
6=1+2+3
28=1+2+4+7+14
496=1+2+4+8+16+31+62+124+248
8128=1+2+4+8+16+32+64+127+254+508+1016+2032+4064
```

例 4-11 对于任意给定的一个正整数 n，统计其阶乘 n!的末尾中 0 的个数。

程序代码如下：

```
using System;

namespace CSHARP4_11
{
    class Program
    {
        static void Main(string[] args)
        {
            int n;
            int sum=0;
            int i,k;
            Console.Write("Pleast input a positive number: ");
            n=Convert.ToInt32(Console.ReadLine());
            for (i=5; i<=n; i=i+5)
            {
                int m=i;
                for (k=0; m%5==0; k++)
                    m=m/5;
                sum=sum+k;
            }
            Console.WriteLine("The number of zero in {0}! is: {1}",n,sum);
        }
    }
}
```

运行结果：

```
Pleast input a positive number: 50
The number of zero in 50! is: 12
```

习题

1. 输入 n(n<13)，计算1!+2!+3!+…+n!。

2. 编写程序求斐波那契数列的第 n 项和前 n 项之和。斐波那契数列形如 0，1，1，2，3，5，8，13，…，其通项为：

$$F_0=0;$$
$$F_1=1;$$
$$F_n=F_{n-1}+F_{n-2}。$$

3. 编程求 $\sin x \approx x-\frac{x^3}{3!}+\frac{x^5}{5!}+\cdots+\frac{(-1)^n x^{2n+1}}{(2n+1)!}+\cdots$，其中 $|x|<1$。

当通项 $|u|<\varepsilon(10^{-7})$ 时，输出运行结果，已知多项式的递推公式为：

$$u_n=\frac{(2n-1)^2 x^2}{2n(2n+1)}u_{n-1}$$

4. 求 π 的近似值。将 arctan(x)在 x=0 展开,得

$$\arctan(x)=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\cdots=\lim_{i=1}(-1)^{i+1}\frac{x^{(2i-1)}}{2i-1}$$

当 x=1 时,arctan(x)=π/4,因此这个级数可以计算 arctan(x)的近似值,还可以计算 π 的近似值:

$$\pi=4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

利用上式编程计算 π 的近似值。

5. 求解猴子吃桃问题。猴子在第一天摘下若干个桃子,当即就吃了一半,又感觉不过瘾,于是就多吃了一个。以后每天如此,到第 10 天时,就只剩下一个桃子了。请编程计算第一天猴子摘的桃子个数。

6. 求 a+aa+aaa+aaaa+…+aa…a(第 n 项,n 个 a),其中 a 是 1~9 的整数。例如,a=1,n=3 时,式子为 1+11+111;当 a=6,n=5 时,式子为 5+55+555+5555+55555。

第5章 数　组

5.1　数组概述

与 C/C++ 语言一样,在 C# 中数组也是广泛使用的一种数据结构。数组是按顺序排列的一组相同类型变量的集合。每个数组用一个数组名标识,数组中每个变量可通过数组名及该变量的下标引用。数组中的变量又称为数组元素,变量的下标又称为索引号或位置序号。默认情况下,数组中第一个元素的下标为 0(零),其后元素的下标依次加 1。这样,具有 n 个元素的数组其下标范围是从 0 到 n－1。数组用于存放多个相同类型的数据,这些数据元素的类型可以是任何值类型(如简单类型、枚举、结构),也可以是引用类型(如 String、数组、类)。

在 C# 中,数组是引用类型,它继承.NET 类库中名为 System.Array 的公共基类,这样可直接使用该基类中定义的各种属性和方法。例如:使用该类的 Length 属性可以获得数组中元素的总数;使用 Rank 属性可以获得数组的维数;使用 GetLength 方法可以获得数组中某个维的长度。数组也遵从先定义后使用的原则,根据需要可定义一维数组、多维数组或交错数组。

5.1.1　声明和创建一维数组

声明和创建一维数组有 4 种格式,分别介绍如下。

格式一:先声明数组,后创建数组,并使用系统默认的初始化值。

```
数组类型 [] 数组名;
数组名=new数组类型 [数组长度];
```

例如:

```
int[] array1;
array1=new int[5];
```

上例声明了含有 5 个元素的一维整型数组 array1。该数组的 5 个元素分别为 array1[0], array1[1], array1[2], array1[3], array1[4],各元素的初值为系统默认值 0。

说明:声明数组时,系统仅定义一个数组变量(即引用数组的变量),并不需要给出数组的长度。声明中的方括号([])必须跟在数组类型的后面,而不是在数组名的后面。new

关键字用于创建数组,因此必须给出数组的长度(初始化时例外),数组长度允许使用常量或变量表达式。系统按照数组元素类型和数组长度在堆中为数组分配存储空间,并使用声明的数组变量引用该数组。

格式二:声明并创建数组。

```
数组类型[] 数组名=new 数组类型[数组长度];
```

例如:

```
int[] array1=new int[5];        //创建含有5个元素的一维整型数组 array1
```

格式三:创建数组并对数组初始化(由用户提供初始值)。

```
数组类型[] 数组名=new 数组类型[数组长度]{初始值列表};
```

其中,初始值列表为用逗号(,)分隔的各初始元素值。

例如:创建一维整型数组 array1,含有5个元素,初始化值依次为1,2,3,4,5。

以下表示都是正确的。

```
int[] array1;
array1=new int[5]{1,2,3,4,5};
```

或

```
int[] array1=new int[5]{1,2,3,4,5};
```

或

```
int[] array1=new int[]{1,2,3,4,5};     //提供初始值列表后,可以省略数组长度。
```

或更简洁的方式:

```
int[] array1={1,2,3,4,5};
```

格式四:创建数组长度为变量表达式的数组。

例如:创建n个元素的一维字符串数组 str。

```
//键盘输入 n
int n=Convert.ToInt32(Console.ReadLine());
//创建 n个元素的一维字符串数组 str
string[] str=new string[n];
```

其中,str 数组元素为引用变量(每个元素可引用一个字符串),系统提供的默认初值为 null。

5.1.2 数组元素的访问

1. 一维数组元素的引用

引用形式:

数组名[下标]

其中,下标允许是整型常数、整型变量或整数表达式。

2. 使用循环语句访问数组元素

例如:反序输出数组元素值。

```
int [] array1=new int[]{1,2,3,4,5};
for(int i=array1.Length-1;i>=0;i--)
Console.Write("{0} ",array1[i]);
Console.WriteLine();
```

输出结果:

```
5 4 3 2 1
```

在访问数组元素时,应防止数组元素下标值越界。例如:

```
for(int i=array1.Length;i>=0;i--)
    Console.Write("{0} ",array1[i]);
```

执行时将产生访问数组越界的错误。因为 array1.Length 表示数组 array1 中总的元素个数,该数组中最后一个元素的下标应为总的元素个数－1。

5.1.3 数组使用举例

例 5-1 求一维数组中的最大值。

假设一维数组元素值为:87,65,78,90,57 和 89。

程序代码如下:

```
//求数组中的最大值
using System;
namespace CSHARP5_1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] a=new int[]{87,65,78,90,57,89};
            int max=a[0];
            for(int i=1;i<a.Length;i++)
                if(a[i]>max)
                    max=a[i];
            Console.WriteLine("The largest number is {0}",max);
        }
    }
}
```

输入和输出：

```
The largest number is 90
```

例 5-2 使用一维数组来求斐波那契数列的第 n 项和前 n 项之和(注：n 从键盘输入，n≥0)，斐波那契数列是这样一个数列：

0，1，1，2，3，5，8，13，…

其通项为：

$$F_0=0;$$
$$F_1=1;$$
$$F_n=F_{n-1}+F_{n-2}\ (n\geq 2)$$

程序代码如下：

```
//求斐波那契数列的第 n项和前 n项之和
using System;
namespace CSHARP5_2
{
    class Program
    {
        static void Main(string[] args)
        {
            int n,sum;
            Console.Write("请输入 n: ");
                n=Convert.ToInt32(Console.ReadLine());
            if(n<0)
                Console.WriteLine("输入错误,n应为大于等于 0的整数!");
            else if(n==0)
                Console.WriteLine("第 0项是 0,和值是 0");
            else if(n==1)
                Console.WriteLine("第 1项是 1,和值是 1");
            else
            { int[] fib=new int[n+1];
                fib[0]=0;
                fib[1]=1;
                sum=1;
                for(int i=2;i<n+1;i++)
                { fib[i]=fib[i-1]+fib[i-2];
                    sum+=fib[i];
                }
                Console.WriteLine("第{0}项是{1},和值是{2}",n,fib[n],sum);
            }
        }
    }
}
```

输入和输出：

```
请输入 n: 10
第 10 项是 55,和值是 143
```

程序代码说明：

第 12 行语句 int[] fib=new int[n+1];创建一维数组 fib,数组长度为 n+1。第 19～26 行计算数列前 n 项的和值,并依次存放在 fib 数组中。设定 fib[0]元素为 0,fib[1]元素为 1,使用 for 语句计算下标从 2 开始的各元素值。

5.1.4 案例研究：洗牌与发牌模拟

例 5-3 开发一个洗牌和发牌的模拟程序,它是编写某种玩牌游戏程序的基础。

设计思想：

声明一个扑克牌类 Card,包含一个一维数组 deck(模拟一副牌,有 52 个元素);一个构造方法(提供 deck 数组的初始信息);洗牌方法 shuffle 和发牌方法 deal。在主方法类中实现对 Card 类的应用处理。

一副牌有 52 张,每张牌包含花色和面值两个数据项。花色分为"红桃"、"方块"、"梅花"和"黑桃"共四种;面值包括"A"、"2"、"3"、"4"、"5"、"6"、"7"、"8"、"9"、"10"、"J"、"Q"和"K"共 13 种。每张牌的信息使用结构类型 CNode 定义。

在 Card 类的构造方法中,完成对模拟牌的数组 deck 初始化。其方法为：将 52 张牌分成 4 组,每组花色相同,面值依次为"A"、"2"～"10"、"J"、"Q"、"K"共 13 个。初始化完成后,数组 deck 中每张牌的排列顺序如图 5-1 所示。

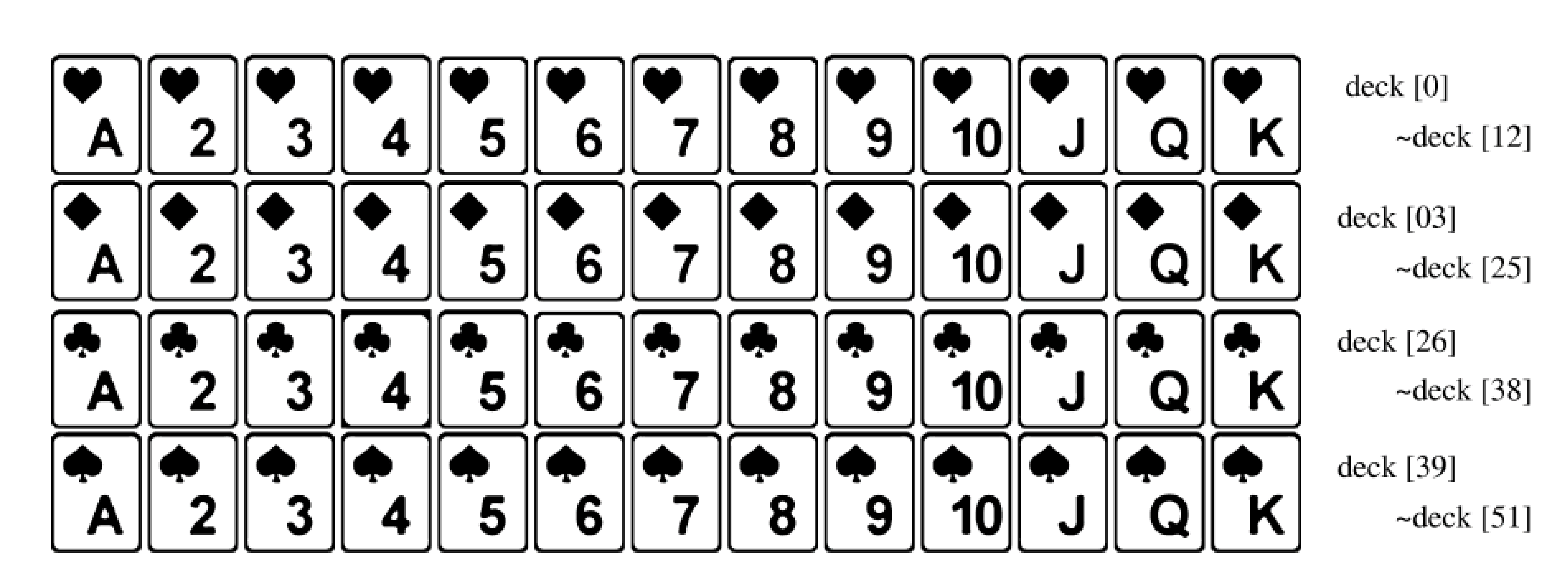


图 5-1 deck 数组中牌的初始顺序

洗牌操作算法描述：

对于数组 deck 的每一张牌 deck[i](i 从 0 到 51 变化),每次随机摘取 0～51 中的一个数 j,然后,对换 deck[i]与 deck[j]元素。经过 52 次对换,整个数组便完成了一遍对调,这样数组中的牌便洗好了。洗牌之后,数组 deck 中牌的顺序如图 5-2 所示。

在洗牌过程中,使用类 Random 的方法 next 产生随机数。算法描述如下：

```
选取系统当前时间作为随机种子;
for (i=0;i<牌的张数;i++)
```

```
{
    生成下一个随机数给 j 变量 (j 值不会超过牌的张数);
    将 i 号牌与 j 号牌对调;
}
```

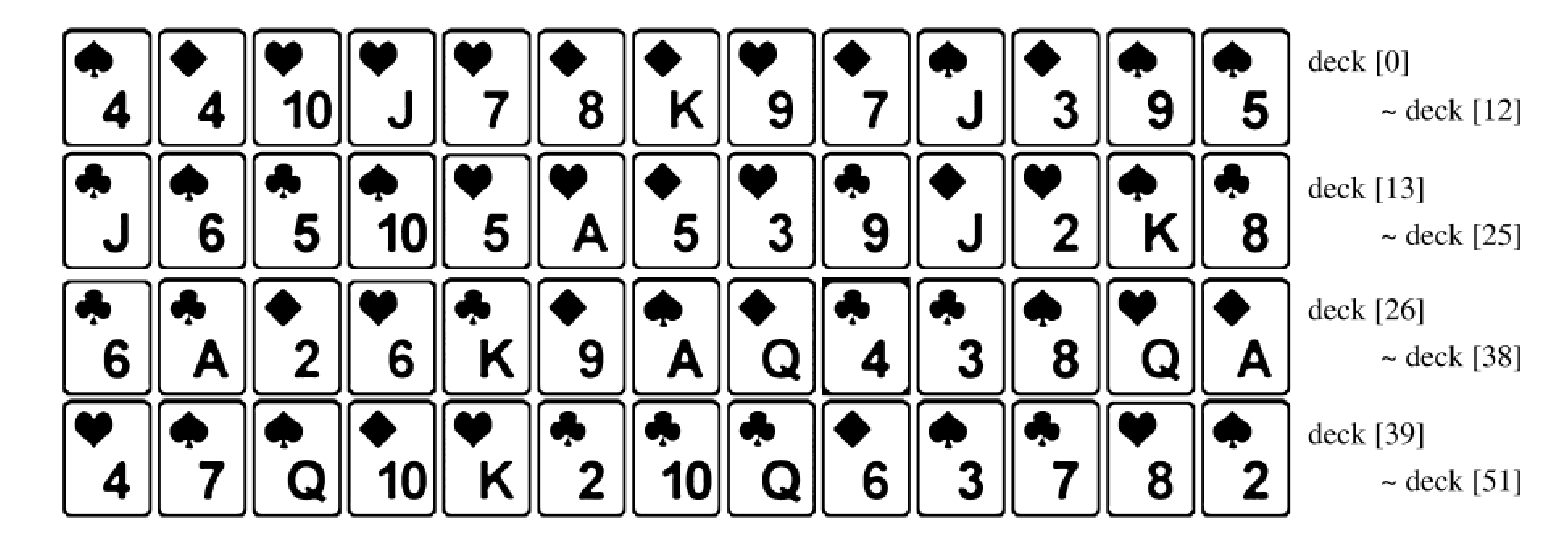


图 5-2　洗牌后 deck 中牌的顺序

发牌操作算法描述：

假设有甲、乙、丙和丁四人玩牌。将洗好的 52 张牌从 deck[0]到 deck[51]依次轮流发送给甲、乙、丙和丁四人。如将 deck[0]发给甲，deck[1]发给乙，deck[2]发给丙，deck[3]发给丁，deck[4]发给甲，deck[5]发给乙，deck[6]发给丙，deck[7]发给丁……重复进行下去，最终每人将得到 13 张牌。图 5-3 是为甲、乙、丙和丁四人所分发的牌。

图 5-3　为甲、乙、丙、丁四人所分发的牌

程序代码如下：

```
//洗牌和发牌模拟
using System;
namespace CSHARP5_3
{
    struct CNode                    //扑克牌的信息结构
    {
        public char suit;           //花色
        public string face;         //面值
    }
    class Card                      //扑克类
```

```
{
    enum Number{CARD_NUMBER=52,SUIT_NUMBER=4,FACE_NUMBER=13 };
                                          //三个枚举常量定义
    private CNode[] deck=new CNode[(int)Number.CARD_NUMBER];
                                          //一副扑克牌
    public Card()                         //构造方法
    {
        char[] suit={ (char)3,(char)4,(char)5,(char)6 };
                                          //"红桃","方块","梅花","黑桃"
        string[] face={"A","2","3","4","5","6","7","8","9","10","J","Q","K"};

        for (int i=0;i<(int)Number.CARD_NUMBER;i++)
        {
            deck[i].face=face[i%(int)Number.FACE_NUMBER];
            deck[i].suit=suit[i/(int)Number.FACE_NUMBER];
        }
    }

    public void shuffle()                 //洗牌方法
    {
        int j;
        CNode temp;
        Random r=new Random();            //系统自动选取当前时间作为随机种子
        for (int i=0;i<(int)Number.CARD_NUMBER;i++)
        {
            j =r.Next()%(int)Number.CARD_NUMBER;
            temp=deck[i];
            deck[i]=deck[j];
            deck[j]=temp;
        }
    }
    public void deal()                    //发牌方法
    {
        Console.Write ("========================");
        Console.Write("52张牌发牌次序 ");
        Console.WriteLine("========================");
        Console.WriteLine("      甲          乙          丙          丁");
        for (int i=0;i<(int)Number.CARD_NUMBER;i++)
        {
            Console.Write ("第{0,2}张:{1}{2}\t",i+1,deck[i].suit,deck[i].face);
            if ((i+1)%4==0)
            {
                Console.WriteLine();
            }
        }
        Console.WriteLine();
```

```
54:             }
55:         }
56:         class Program                           //主类
57:         {
58:             static void Main(string[] args)
59:             {
60:                 Card obj=new Card();            //创建一副扑克牌
61:                 obj.shuffle();                  //洗牌
62:                 obj.deal();                     //发牌
63:                 Console.ReadKey();
64:             }
65:         }
66: }
```

程序执行结果如图 5-4 所示。

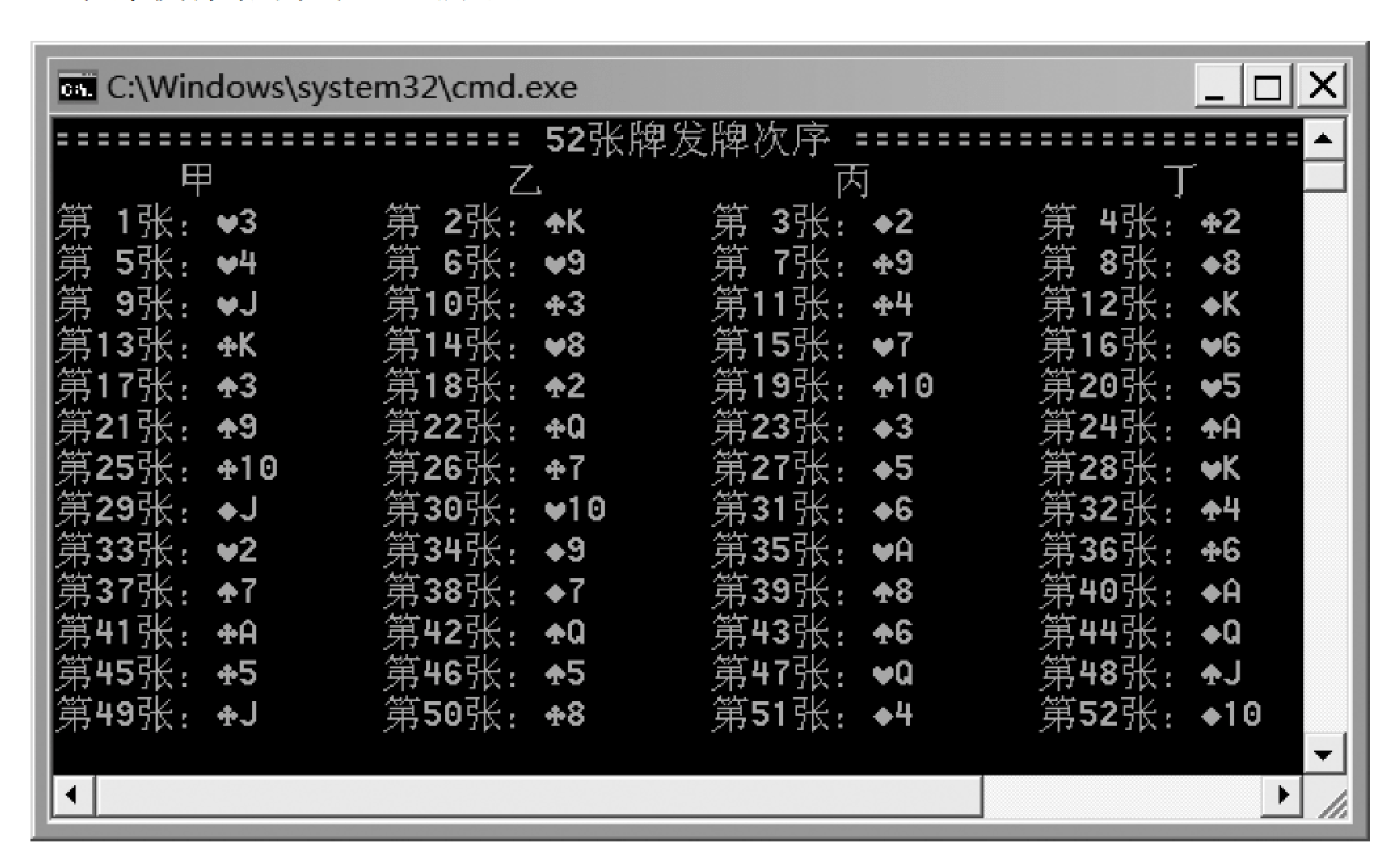


图 5-4 洗牌和发牌模拟程序的执行结果

程序说明：

一副扑克有 52 张牌，每张牌包含花色和面值两项信息。程序的第 5～9 行定义一张扑克牌的信息结构，类型名为 CNode。

第 10～24 行定义扑克牌类 Card。其中，第 12 行的 Number 为枚举类型，含有 3 个枚举常数。CARD_NUMBER 表示整数 52，代表 52 张牌；SUIT_NUMBER 表示整数 4，代表 4 种花色；FACE_NUMBER 表示整数 13，代表一副扑克牌有 13 种面值。

第 13 行创建一维数组 deck，用于模拟一副 52 张的扑克牌。该数组元素个数为 52，使用枚举常量 CARD_NUMBER 表示，元素类型为 CNode。

第 14～23 行为构造方法，完成扑克牌数组 deck 的初始化。其中，第 16 行定义花色

数组 suit，其初值 3、4、5 和 6 为四种花色字符的 Unicode 编码，而(char)3、(char)4、(char)5 和(char)6 分别代表♥"红桃"、♦"方块"、♣"梅花"和♠"黑桃"四种花色字符。第 17 行定义扑克牌的面值数组 face，其初值为"A"、"2"、"3"、"4"、"5"、"6"、"7"、"8"、"9"、"10"、"J"、"Q"和"K"共 13 个字符串数据。

第 26～38 行为洗牌方法。其中，第 30 行定义了产生随机数的对象变量 r，并采用系统时间值作为随机数发生器函数的种子。第 33 行是为 j 变量产生 0～51 之间的一个随机整数，而 r.Next()可以产生 0～2147483647 之间的一个整数。

第 39～54 行为发牌方法。其中，第 47 行中的占位符{0,2}，其意义为输出项值占 2 位宽度，不足 2 位左边添一个空格。

例如：

当 i=0,47 行语句执行后，输出：第 1 张：♥3(第 1 张为红桃 3)。

当 i=9,47 行语句执行后，输出：第 10 张：♣3(第 10 张为梅花 3)。

5.2 foreach 语句

foreach 是 C# 语言新增的一个循环语句，专用于访问数组和集合中的元素。该语句结构简洁，执行效率更高。

foreach 语句的格式：

```
foreach(类型 循环变量 in 数组名或集合对象)
{
    循环体
}
```

其中，类型为循环变量的类型，通常情况下，该类型与被访问的数组或集合元素类型相同(也可使用 var 声明通用型的循环变量)。

在 foreach 循环中，通过循环变量访问数组或集合中的每个元素。首先，让循环变量获取数组或集合中的第一个元素值，执行循环体后，该变量将顺序获取下一个元素值，继续执行循环体……直到数组或集合中的所有元素被访问。当最后一个元素被访问后，控制将传递给 foreach 语句块之后的下一条语句继续执行程序。

与 for、while 等循环语句一样，在 foreach 语句中也可以使用 break 语句跳出循环，或使用 continue 进入下一次循环。

例 5-4 使用 foreach 显示整型数组的内容。

程序代码如下：

```
//使用 foreach 显示整型数组
using System;
namespace CSHARP5_4
{
    class Program
    {
```

```
07:         static void Main(string[] args)
08:         {
09:             int[] array1=new int[] {1,2,3,4,5};     //array1 为整型数组
10:             foreach (int i in array1)               //i 为循环变量,是整型
11:             {
12:                 Console.WriteLine(i);
13:             }
14:         }
15:     }
16: }
```

程序执行结果如下：

```
1
2
3
4
5
```

程序说明：

第 10～13 行为 foreach 语句块,用于显示数组 array1。第 10 行中 i 为循环变量,其类型声明为 int,该类型与数组 array1 元素类型相同。

5.3 数组的参数传递

5.3.1 将数组和数组元素传入方法

数组和数组元素均可作为参数传入方法(函数)。在 C# 中,数组属于引用类型,数组作实参传递的是数组的引用(即数组在内存中的地址)。

数组作为实参,仅提供不带方括号的数组名,而形参必须声明为同类型的数组。调用时,向方法传递数组的引用(此时,形参作为数组引用的“副本”),方法通过形参间接访问实参数组。因此,方法中改变形参数组的值也就同时改变了实参数组的值。

数组元素作为实参传入方法的是该元素值的“副本”,方法中对“副本”进行操作,而“副本”值的任何修改都不会改变实参元素的值。

例 5-5 数组作为参数。

使用一维数组作为方法的参数,对数组进行逆置操作。

程序代码如下：

```
using System;
namespace CSHARP5_5
{
    class Program
    {
        static void Revers(int[] arr)     //一维数组 arr 作为方法参数
```

```
07:         {
08:             for (int i=0; i<arr.Length/2; i++)
09:             {
10:                 int temp;
11:                 temp=arr[i];
12:                 arr[i]=arr[arr.Length - i - 1];
13:                 arr[arr.Length - i - 1]=temp;
14:             }
15:         }
16:         static void PrintArr(int[] arr)   //一维数组 arr 作为方法参数
17:         {
18:             foreach(int x in arr)
19:                 Console.Write("{0} ",x);
20:             Console.WriteLine();
21:         }
22:
23:         static void Main(string[] args)
24:         {
25:             int[] a={1,2,3,4,5};
26:             Console.WriteLine("数组的初值:");
27:             PrintArr(a);                  //数组 a 作为实参
28:             Revers(a);                    //数组 a 作为实参,调用 Revers 方法逆置数组 a 的值
29:             Console.WriteLine("逆置后的数组值:");
30:             PrintArr(a);                  //数组 a 作为实参
31:         }
32:     }
33: }
```

程序代码说明：

主类 Program 包括 3 个方法，除主方法(第 23～31 行)外，方法 Revers(第 6～15 行)实现数组的逆置，而方法 PrintArr(第 16～21 行)实现数组的显示。

在主方法中，第 18 行调用 Revers 时的实参为数组名 a，而 Revers 方法中形参 arr 接受的是 a 数组的引用(即 a 数组的内存首地址)。从表面上看，Revers 方法中访问 arr 数组，其实际是通过 arr 间接访问对应的实参 a 数组。这样，在 Revers 方法中对 arr 数组逆置操作后，主方法中实参数组 a 的值也就被逆置了。

程序执行结果如下：

```
数组的初值:
1 2 3 4 5
逆置后的数组值:
5 4 3 2 1
```

例 5-6 数组元素作为参数。

阅读程序，分析程序执行的结果(注：该程序不能实现对一维数组的逆置操作)。

程序代码如下：

```
01: using System;
02: namespace CSHARP5_6
03: {
04:        class Program
05:        {
06:             static void Swap(int x,int y)    //方法 Swap 互换 x 与 y
07:             {
08:                  int temp;
09:                  temp= x;
10:                  x= y;
11:                  y= temp;
12:             }
13:             static void PrintArr(int[] arr)
14:             {
15:                  foreach(int x in arr)
16:                       Console.Write("{0} ",x);
17:                  Console.WriteLine();
18:             }
19:             static void Main(string[] args)
20:             {
21:                  int[] a= {1,2,3,4,5};
22:                  Console.WriteLine("数组的初值:");
23:                  PrintArr(a);            //数组 a 作为实参
24:                  Swap(a[0],a[4]);        //数组元素作为参数,互换 a[0]和 a[4]元素的值
25:                  Swap(a[1],a[3]);        //数组元素作为参数,互换 a[1]和 a[3]元素的值
26:                  Console.WriteLine("互换 a[0]和 a[4],a[1]和 a[3]后,a数组值:");
27:                  PrintArr(a);            //数组 a 作为实参
28:             }
29:        }
30: }
```

程序代码说明：

主类 Program 包括 3 个方法，除主方法(第 19～28 行)外，方法 Swap(第 6～12 行)用于互换两个形参变量的值，而方法 PrintArr(第 16～21 行)实现数组的显示。

程序运行结果：

```
数组的初值:
1 2 3 4 5
互换 a[0]和 a[4],a[1]和 a[3]后,a数组值:
1 2 3 4 5
```

从运行结果看，主方法在调用 Swap 前、后的第 23 行和第 27 行两次输出 a 数组，其输出结果完全相同。为什么在执行 24 和 25 行后 a 数组的值没有交换？

在 C# 中，值类型变量或数组元素作为实参，传递给方法的是该值的副本（这时，实参与形参具有相互独立的存储单元，并且只能在各自声明的方法中使用），方法中使用和修改的是副本的值，而对应实参的原变量值是不会改变的，这一点 C# 与 C/C++ 语言的处理是相同的。

执行主方法中第 24 行的方法调用时，系统将实参 a[0]和 a[4]数组元素的值传递给 Swap 方法的形参 x 和 y（这时，x 和 y 在栈中分配有独立的存储单元），在方法中完成 x 与 y 的互换，而实参 a[0]与 a[4]值保持不变。同样，执行第 25 行后，实参 a[1]与 a[3]的值也不会改变。因此，数组元素作为参数，方法中不会改变实参的值。

5.3.2　案例研究：GradeBook 类用数组保存成绩

例题综述：

学生考试成绩管理（供教师管理学生考试成绩的成绩簿）由成绩簿类（GradeBook）和主方法类（Program）构成。

GradeBook 类包含的字段成员有学生人数（STUDENTS）、课程名（courseName）和引用一门课程成绩的数组变量（grades）以及多个操作方法，如课程名的属性 CourseName、初始化 GradeBook 类对象的构造方法、显示课程名的方法以及各种用于数据管理的方法，如：①寻找最低成绩；②寻找最高成绩；④计算平均成绩；⑤打印条形统计图表；⑥打印成绩表；⑦处理成绩汇总。

GradeBook 类结构描述如下：

```
class GradeBook
{
    字段成员：
    学生人数 (STUDENTS)
    课程名 (string courseName)
    成绩数组 (int [] grades)
    成员方法：
    课程名的属性 - CourseName
    构造方法 - GradeBook(string name,int[] gradeArray)
    显示课程名 - displayMessage()
    最低成绩 - getMinimum()
    最高成绩 - getMaximum()
    平均成绩 - getAverage()
    打印条形统计图表 - outputBarChart()
    打印成绩表 - outputGrades()
    成绩汇总 - processGrades()
}
```

定义主方法类（Program），并在主方法中创建 GradeBook 对象，通过该对象实现对某门课程考试成绩的管理应用。

主方法类描述如下：

```
class Program
{
    static void Main(string[] args)
    {
        定义课程名变量并赋以初值;
        定义一维数组并初始化为学生成绩数据;
        创建 GradeBook 对象并进行初始化(利用带参数的构造方法);
        通过对象访问"成绩汇总"方法。
    }
}
```

例 5-7

程序代码如下:

```
//学生成绩汇总
using System;
namespace CSHARP5_7
{
    class GradeBook                                  //成绩簿类
    {
        private readonly int STUDENTS;               //只读变量,学生人数
        private string courseName;                   //课程名称
        private int[] grades;                        //引用成绩数组的变量
        public GradeBook(string name,int[] gradeArray)                      //构造方法
        {
            STUDENTS=gradeArray.Length;              //初始化只读变量
            CourseName=name;
            grades=gradeArray;                       //grades 将引用实际数组
        }
        public string CourseName                     //课程名字段的属性
        {
            get
            {
                return courseName;
            }
            set
            {
                courseName=value;
            }
        }
        public void displayMessage()
        {
            Console.WriteLine("《{0}》成绩汇总",CourseName);                //引用课程名属性
        }
        public int getMinimum()                      //最低成绩
        {
```

```
        int lowGrade=100;                  //假设最低成绩是 100
        foreach(int grade in grades)
            if (grade< lowGrade)
                lowGrade=grade;
        return lowGrade;
    }
    public int getMaximum()                //最高成绩
    {
        int highGrade=0;                   //假设最高成绩是 0
        foreach(int grade in grades)
            if (grade > highGrade)
                highGrade=grade;
        return highGrade;
    }
    public double getAverage()             //平均成绩
    {
        int total=0;
        foreach (int grade in grades)
            total+=grade;
        return (double)total/STUDENTS;
    }
    public void outputBarChart()           //打印条形统计图
    {
        Console.WriteLine("\n成绩分布表：");
        int[] frequency=new int[11];
        foreach (int grade in grades)
            frequency[grade/10]++;
        for (int i=0; i<frequency.Length; i++)
        {
            if (i==0)
                Console.Write("  0~9: ");
            else if (i==10)
                Console.Write("  100: ");
            else
                Console.Write(i*10+"-"+(i*10+9)+": ");
            for (int j=0; j<frequency[i]; j++)
                Console.Write("*");
            Console.WriteLine();
        }
    }
    public void outputGrades()             //打印成绩表
    {
        Console.WriteLine("\n成绩是：\n");
        for (int i=0; i<STUDENTS; i++)
            Console.WriteLine("学生 {0,2}: {1,3}",i+1,grades[i]);
        Console.WriteLine();
```

```
79:         }
80:         public void processGrades()                    //成绩汇总
81:         {
82:             outputGrades();                            //输出成绩表
83:             Console.WriteLine("课程平均值是 {0:f2}",getAverage());
                                                           //输出课程平均值
84:             Console.WriteLine("最低成绩是 {0}\n最高成绩是 {1}",getMinimum(),
                    getMaximum());                         //输出最低和最高成绩
85:             outputBarChart();                          //输出条形统计图
86:         }
87:     }
88:     class Program
89:     {
90:         static void Main(string[] args)
91:         {
92:             string courseName=" C# 程序设计";          //定义课程名
93:             int[] gradesArray=new int[] { 87,68,94,100,83,78,85,91,76,87};
                                                           //定义成绩数组并初始化
94:             GradeBook myGradeBook=new GradeBook(courseName,gradesArray);
                                                           //创建成绩簿对象
95:             myGradeBook.displayMessage();              //显示课程信息
96:             myGradeBook.processGrades();               //调用成绩汇总
97:         }
98:     }
99: }
```

在 GradeBook 类(05～87 行)中,第 7 行声明一个只读(readonly)变量 STUDENTS,表示学生人数,其初值可由构造函数提供(不同对象可以有不同的初值),之后该变量值保持不变(相当于一个常量);而 const 定义的常量在编译阶段完成,所有对象具有相同的常量值。第 9 行声明的是引用一维数组的变量 grades。10～15 行为构造函数,其中第 2 个参数为引用一维数组的引用变量。16～26 行为课程名字段 courseName 定义的属性。27～30 行定义方法 displayMessage,显示课程名。31～38 行定义方法 getMinimum,计算最低成绩。39～46 行定义方法 getMaximum,计算最高成绩。47～53 行定义方法 getAverage,计算平均成绩。54～72 行定义方法 outputBarChart,打印条形统计图表。73～79 行定义方法 outputGrades,打印成绩表。80～86 行定义方法 processGrades,处理成绩汇总。90～97 行为主方法,完成对课程名为“C＃程序设计”的一组成绩进行汇总。其中第 92 行定义课程名变量 courseName,并初始化“C＃程序设计”。第 93 行定义一维数组 gradesArray 并初始化为一组成绩数据(成绩个数任意,本例提供 10 个)。第 94 行创建类 GradeBook 的对象 myGradeBook,成功创建后,myGradeBook. courseName 将引用实参 courseName,myGradeBook. grades 将引用实参数组 gradesArray,而 myGradeBook.STUDENTS 的初始值为 gradesArray. Length 的当前值(本例中值为 10)。第 95 行将显示对象 myGradeBook 的课程信息。第 96 行调用对象 myGradeBook 的方法 processGrades 实现成绩汇总。

程序运行结果如图 5-5 所示。

```
《C#程序设计》成绩汇总

成绩是:

学生  1:  87
学生  2:  68
学生  3:  94
学生  4: 100
学生  5:  83
学生  6:  78
学生  7:  85
学生  8:  91
学生  9:  76
学生 10:  87

课程平均值是 84.90
最低成绩是 68
最高成绩是 100

成绩分布表:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
请按任意键继续. . .
```

图 5-5　程序运行结果

5.4　多维数组

C# 支持多维数组(二维及以上维数)。数组类型的维数也称为数组类型的秩,它是数组类型的方括号之间的逗号个数加上 1。

5.4.1　多维数组的使用

1. 多维数组的声明与初始化

二维数组即数组的维数为 2。二维数组可以表示矩阵,它有两种格式。

格式一:声明并创建二维数组,初始化为系统的默认值。

```
数组类型 [,] 数组名;
数组名=new 数组类型 [m,n];
```

或

```
数组类型 [,] 数组名=new 数组类型 [m,n];
```

其中,m,n 可以是常量或变量表达式。

注意：C#中二维数组类型的标志“数组类型[,]”，不是C/C++中的“数组名[][]”。

例如，声明并创建一个四行两列的二维数组。

```
int[,] array2;
arra2=new int[4,2];
```

或

```
int[,] array2=new int[4,2];
```

其中，array2数组包含4×2共8个元素，元素的初始值默认为0。

格式二：创建二维数组，并提供初始化值。

```
数组类型 [,] 数组名=new 数组类型 [,]{{…},{…},…,{…}};
```

或

```
数组类型 [,] 数组名={{…},{…},…,{…}};
```

例如，创建成绩二维数组score，并初始化下列元素值。

86 92 78 99 87
78 87 88 90 77
79 80 87 67 89

```
int [,] score=new int[3,5]{{ 86,92,78,99,87},{78,87,88,90,77},{79,80,87,67,89}};
```

或

```
int [,] score=new int[,]{{ 86,92,78,99,87},{78,87,88,90,77},{79,80,87,67,89}};
```

或

```
int [,] score={{ 86,92,78,99,87},{78,87,88,90,77},{79,80,87,67,89}};
```

三维数组即数组的维数为3。例如：声明并创建一个三维数组：

```
int[,,] array3=new int[4,2,3];
```

其中array3是一个三维数组，包含4×2×3共24个元素。

2. 交错数组（数组的数组）

交错数组是元素为数组的数组。交错数组元素的长度可以不同。利用交错数组可以更有效地使用内存空间。

例如：声明并创建一个由三个元素组成的一维数组，其中每个元素都是一个一维整数数组，且第一个数组元素有5个元素，第二个数组元素有4个元素，第三个数组元素有2个元素。

```
int[][] jagArray=new int[3][];        //交错数组声明
jagArray[0]=new int[5];
jagArray[1]=new int[4];
```

```
jagArray[2]=new int[2];
```

说明：

(1) jagArray 是一个交错数组，包含 3 个元素 jagArray[0]、jagArray[1]和 jagArray[2]，其中 jagArray[i]是一个整型数组变量，属于引用型变量，默认的初值是 null。

(2) jagArray[i]代表第 i 个一维整型数组。

如 jagArray[0]数组包含 5 个元素，分别是：jagArray[0][0]，jagArray[0][1]，jagArray[0][2]，jagArray[0][3]，jagArray[0][4]，该整型数组元素默认的初值为 0。

也可对交错数组进行初始化操作，这时可省略数组的长度。例如：

```
int[][] jagArray=new int[3][];          //交错数组声明
jagArray[0]=new int[] { 1,3,5,7,9 };
jagArray[1]=new int[] { 0,2,4,6 };
jagArray[2]=new int[] { 11,22 };
```

3. 多维数组的使用

例 5-8 编写一个程序，实现矩阵加法运算。

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

矩阵加法原理：

设有 M 行 N 列的矩阵，则由线性代数得知，其和仍为 M 行 N 列的矩阵。

算法：用两重循环实现求和。

```
for(i=0;i<M;i++)
  for(j=0j<N;j++)
    c[i,j]=a[i,j]+b[i,j];
```

程序代码如下：

```
//CSHARP5_8 矩阵加法
using System;
namespace CSHARP5_8
{
      class Program
      { const int M=3,N=4;
           static void Main(string[] args)
           {
                int[,] a=new int[,]{{ 1,2,3,4},{5,6,7,8},{9,10,11,12}};
                int[,] b=new int[,]{{ 1,4,7,10},{2,5,8,11},{3,6,9,12}};
                int[,] c=new int[M,N];
                //矩阵求和
                for (int i=0; i<M; i++)
```

```
                for (int j=0; j<N; j++)
                    c[i,j]=a[i,j]+b[i,j];
            //按 M行 N列输出矩阵 c,每个数据占 4位宽度
            Console.WriteLine("结果矩阵是：");
            for (int i=0; i<c.GetLength(0); i++)
            {
                for (int j=0; j<c.GetLength(1); j++)
                    Console.Write("{0,4}",c[i,j]);
                Console.WriteLine();
            }
        }
    }
}
```

程序运行结果如图 5-6 所示。

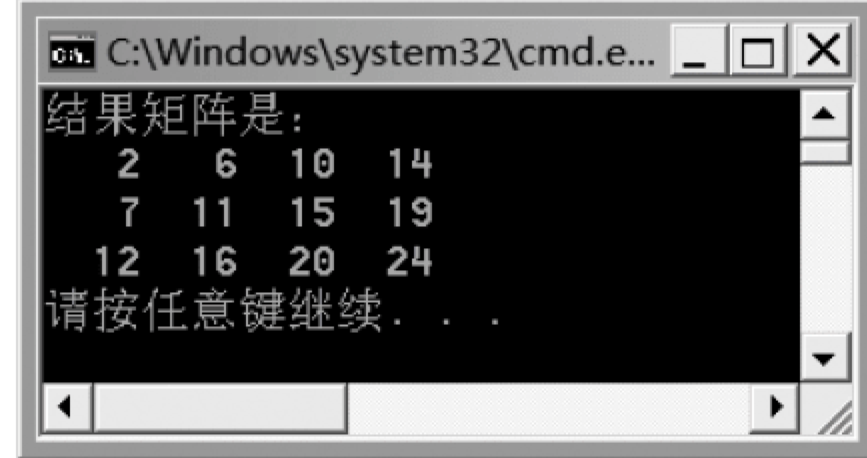


图 5-6 矩阵加法

例 5-9 利用交错数组生成如下杨辉三角(又称 Pascal 三角)。

```
1
1  1
1  2   1
1  3   3   1
1  4   6   4    1
1  5  10  10    5    1
1  6  15  20   15    6   1
1  7  21  35   35   21   7   1
1  8  28  56   70   56  28   8  1
1  9  36  84  126  126  84  36  9  1
```

编程描述：

(1) 声明一个交错数组 yh,这是由 10 个元素组成的一维数组,其中每个元素又是由不等长的一维整型数组组成。

```
int [][] yh=new int[10][];      //声明并创建 10个元素的交错数组
for(int i=0;i<10;i++)
  yh[i]=new int[i+1];           //分别创建 10个一维数组,其中 yh[i]为具有 i+1个元素的一维整型数组
```

(2) 第 i 行各项值的变化规律(i 从 1 开始)：

```
yh[i][0]=1;                          //第 0列元素为 1
yh[i][i]=1;                          //最后一列元素为 1
for(j=1;j<i;j++)
  yh[i][j]=yh[i-1][j-1]+yh[i-1][j];  //其余各列元素值
```

程序代码如下：

```
//CSHARP5_9计算杨辉三角
using System;
namespace CSHARP5_9
{
    class Program
    {
        static void Main(string[] args)
        {
            int[][] yh=new int[10][];
            for (int i=0; i<10; i++)
                yh[i]=new int[i+1];
            yh[0][0]=1;
            for (int i=1; i<10; i++)
            {
                yh[i][0]=1;                //第 0列元素为 1
                yh[i][i]=1;                //最后一列元素为 1
                for (int j=1; j<i; j++)
                    yh[i][j]=yh[i-1][j-1]+yh[i-1][j];  //其余各列元素值
            }
            //输出杨辉三角
            Console.WriteLine("杨辉三角输出结果:");
            for (int i=0; i<10; i++)
            {
                for (int j=0; j <=i; j++)
                    Console.Write("{0,6}",yh[i][j]);
                Console.WriteLine();
            }
        }
    }
}
```

程序运行结果如图 5-7 所示。

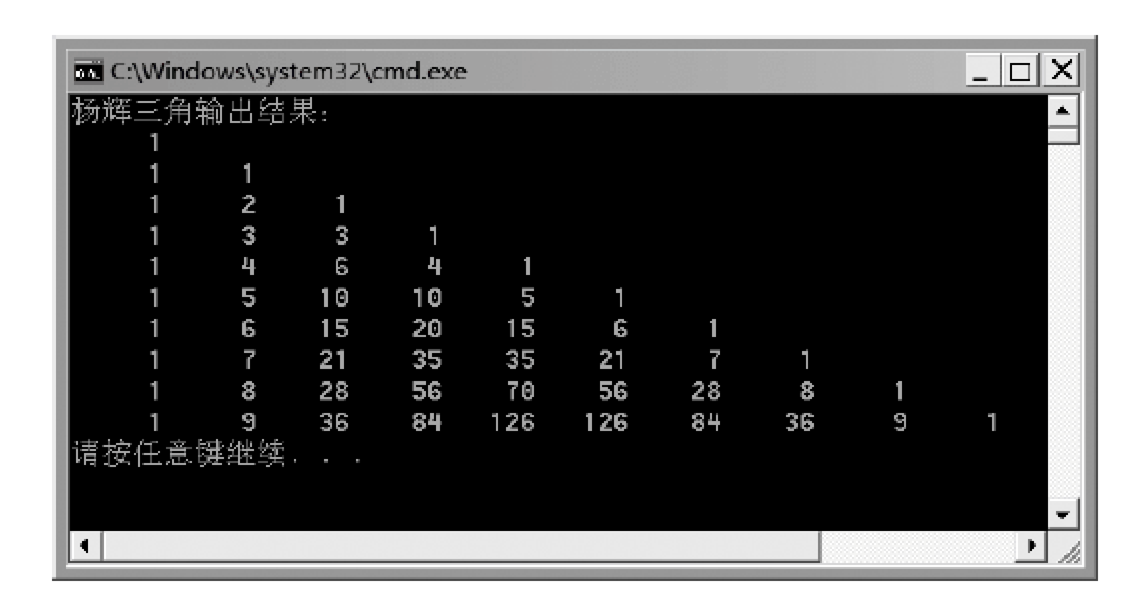


图 5-7　计算杨辉三角

5.4.2 案例研究：使用矩形数组的 GradeBook

在 5.3.2 节的案例程序 CSHARP5_7 中，使用一个一维数组存储学生在一次考试中所得的成绩。在一个学期中，学生很可能参加多次考试，教师也希望分析一个学生或整个班级在整个学期所得的成绩，那么可以对 CSHARP5_7 进行改动。在 GradeBook 类中使用矩形数组 grades 存储多名学生在多次考试中获得的考试成绩。该数组的每一行表示一个学生参加的一门课的多次考试成绩，每一列表示参加某次考试的所有学生的成绩。假设有 10 名学生参加了一门课程的 3 次考试，所有成绩使用 10×3 的二维数组存储，则 GradeBook 类改动如下：

```
class GradeBook
{
    字段成员：
    学生人数 (STUDENTS)
    测试次数 (TESTS)
    课程名 (string courseName)
    成绩数组 (int [,] grades)        //二维矩形数组
    成员方法：
    课程名的属性：CourseName
    构造方法：GradeBook(string name,int[,] gradeArray)
    显示课程名：displayMessage()
    最低成绩：getMinimum()
    最高成绩：getMaximum()
    平均成绩：getAverage(int)
    打印条形统计图表：outputBarChart()
    打印成绩表：outputGrades()
    成绩汇总：processGrades()
}
```

例 5-10

程序代码如下：

```
using System;
namespace CSHARP5_10
{
    class GradeBook                                 //成绩簿类
    {
        private readonly int STUDENTS;              //只读变量,学生人数
        private readonly int TESTS;                 //只读变量,测试次数
        private string courseName;                  //课程名称
        private int[,] grades;                      //定义对二维数组(矩阵数组)的引用变量
        public GradeBook(string name,int[,] gradeArray)   //构造方法
        {
```

```
            STUDENTS=gradeArray.GetLength(0);         //获取学生人数
            TESTS=gradeArray.GetLength(1);            //获取测试次数
            CourseName=name;
            grades=gradeArray;
        }
        public string CourseName                      //课程名字段的属性
        {
            get
            {
                return courseName;
            }
            set
            {
                courseName=value;
            }
        }
        public void displayMessage()
        {
            Console.WriteLine("《{0}》成绩汇总",CourseName);
        }
        public int getMinimum()      //最低成绩
        {
            int lowGrade=100;        //假设最低成绩是 100
            foreach (int grade in grades)
            {
                if (grade< lowGrade)
                    lowGrade=grade;
            }
            return lowGrade;
        }
        public int getMaximum()      //最高成绩
        {
            int highGrade=0;         //假设最高成绩是 0
            foreach (int grade in grades)
            {
                if (grade >highGrade)
                    highGrade=grade;
            }
            return highGrade;
        }
        public double getAverage(int i)//计算第 i 个学生的平均成绩
        {
            int total=0;
```

```
            for (int j=0; j<TESTS; j++)
                total+=grades[i,j];
            return (double)total/TESTS;
        }
        public void outputBarChart() //打印条形统计图表
        {
            Console.WriteLine("\n成绩分布表: ");
            int[] frequency=new int[11];
            foreach (int grade in grades)
                frequency[grade/10]++;
            for (int i=0; i<frequency.Length; i++)
            {
                if (i==0)
                    Console.Write(" 0-9: ");
                else if (i==10)
                    Console.Write(" 100: ");
                else
                    Console.Write(i*10+"-"+(i*10+9)+": ");
                for (int j=0; j<frequency[i]; j++)
                    Console.Write("*");
                Console.WriteLine();
            }
        }
        public void outputGrades()   //打印成绩表以及学生的平均分
        {
            Console.WriteLine("\n成绩是: \n");
            Console.Write("\t");
            for (int i=0; i<TESTS; i++)
                Console.Write("测试{0}\t",i+1);
            Console.WriteLine("{0}\t","平均成绩");
            for (int i=0; i<STUDENTS; i++)
            {
                Console.Write("学生{0,2}\t",i+1);
                for (int j=0; j<TESTS; j++)
                    Console.Write("{0}\t",grades[i,j]);
                Console.WriteLine("{0:f2}",getAverage(i));
            }
            Console.WriteLine();
        }
        public void processGrades()  //成绩汇总
        {
            outputGrades();          //输出成绩表
            Console.WriteLine("最低成绩是 {0}\n最高成绩是 {1}",getMinimum(),
                  getMaximum()); //输出最低和最高成绩
            outputBarChart();        //输出条形统计图
```

```
99:             }
100:        }
101:        class Program
102:        {
103:            static void Main(string[] args)
104:            {
105:                int[,] gradeArray= new int[,]{
106:             {87,96,70},
107:             {68,87,90},
108:             {94,100,90},
109:             {100,81,82},
110:             {83,65,85},
111:             {78,87,65},
112:             {85,75,83},
113:             {91,94,100},
114:             {76,72,84},
115:             {87,93,73}};                //定义并初始化二维数组(矩形数组)
116:                GradeBook MyCardeBook= new GradeBook("C# 程序设计",gradeArray);
                                                    //创建成绩簿对象
117:                MyCardeBook.displayMessage();//显示学生信息
118:                MyCardeBook.processGrades();//调用成绩汇总
119:            }
120:        }
121: }
```

在 GradeBook 类(04～100 行)中,第 6 行和第 7 行分别声明 2 个只读(readonly)变量 STUDENTS 和 TESTS,表示学生人数和测试次数,其初值可由构造函数提供(不同对象可以有不同的初值),之后该变量值保持不变(相当于一个常量);第 9 行声明的是引用二维数组的变量 grades。10～16 行为构造函数,其中第 2 个参数为引用二维数组的引用变量。17～27 行为课程名字段 courseName 定义的属性。28～31 行定义方法 displayMessage,显示课程名。32～41 行定义方法 getMinimum,计算最低成绩。42～51 行定义方法 getMaximum,计算最高成绩。52～58 行定义方法 getAverage,计算第 i 个学生的平均成绩。59～77 行定义方法 outputBarChart,打印条形统计图。78～93 行定义方法 outputGrades,打印成绩表以及学生的平均分。94～99 行定义方法 processGrades,处理成绩汇总。103～119 行为主方法,完成对课程名为“C＃程序设计”的 3 次测试成绩进行汇总。其中 105～115 行定义二维数组 gradesArray 并初始化为 10 行 3 列的成绩数据(表示 10 个学生,每个学生有 3 次测试成绩)。第 116 行创建类 GradeBook 的对象 myGradeBook,成功创建后,myGradeBook. courseName 的值为“C＃程序设计”,myGradeBook. grades 将引用实参数组 gradesArray,myGradeBook. STUDENTS 的初始值为 gradeArray. GetLength(0)的当前值(即二维数组的行数),myGradeBook. TESTS 的初始值为 gradeArray. GetLength(1)的当前值(即二维数组的列数)。第 117 行显示对象 myGradeBook 的课程信息。第 118 行调用对象 myGradeBook 的方法 processGrades 实现成绩汇总。

程序运行结果如图 5-8 所示。

```
《C#程序设计》成绩汇总

成绩是:

          测试1   测试2   测试3   平均成绩
学生 1    87      96      70      84.33
学生 2    68      87      90      81.67
学生 3    94      100     90      94.67
学生 4    100     81      82      87.67
学生 5    83      65      85      77.67
学生 6    78      87      65      76.67
学生 7    85      75      83      81.00
学生 8    91      94      100     95.00
学生 9    76      72      84      77.33
学生10    87      93      73      84.33

最低成绩是 65
最高成绩是 100

成绩分布表:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

图 5-8 使用矩形数组的 GradeBook 的程序运行结果

5.5 变长实参表

在方法参数表中,用关键字 params 修饰的数组称为“参数数组”。参数数组对应的实参可使用“变长实参表”。即一个参数数组可接收一个实参(如一个数组名),也可以接收任意多个实参(同类型的数据)。

例 5-11 变长实参表的应用。

程序代码:

```
using System;
namespace CSHARP5_11
{
    class Program
    {
        static double average(params int[] arr)           //形参为参数数组
        {
            int i=arr.GetLength(0);
            double answer=0.0;
            for (int j=0; j<i; j++)
                answer+=arr[j];
            return answer/i;
        }
        public static void Main()
        {
            int[] a={ 1,2,3,6,7 };
            Console.WriteLine("{0}",average(a));          //数组 a 作为实参
            Console.WriteLine("{0}",average(10,2,3));     //3个整数作为实参
            Console.WriteLine("{0}",average(1,1,3,5,7));  //5个整数作为实参
```

```
20:             Console.WriteLine ("{0}",average(a[0],a[1],a[2],a[3],
                                  a[4],8));                    //6个整数表达式作为实参
21:         }
22:     }
23: }
```

程序说明：

代码 06～13 行用于计算“变长实参表”提供的若干数据的平均值。其中第 6 行的方法参数为参数数组，可以接收“变长实参表”。第 17 行的方法调用，实参为数组名 a；而在第 18、19 和 20 行的方法调用中，对应的实参个数分别为 3、5、6。

程序运行结果：

```
3.8
5
3.4
4.5
```

需要说明的是，方法中的“参数数组”只能有一个，只适合一维数组（如 arr[]和 arr[][]都可以，而 arr[,]不可以）且必须是形参列表中的最后一个参数。

5.6　使用命令行实参

与 C/C++ 一样，在 C# 中 Main 方法也能接受命令行实参（即由命令行提供的参数）。这时，Main 方法须带有 string[]类型的形参（即字符串数组），用于存放命令行提供的各个实际参数，各实参之间须用空格分隔。与 C/C++ 不同的是，C# 不将执行的程序名称（即命令名）视为命令行的第一个实参。

例 5-12　访问来自命令行的实参。

程序代码如下：

```
using System;
namespace CSHARP5_12
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("参数个数：{0}",args.Length);
            for (int i=0; i<args.Length; i++)
            {
                Console.WriteLine("Arg[{0}]=[{1}]",i,args[i]);
            }
        }
    }
}
```

通过命令行提供实参的命令格式：

命令名 参数 1 参数 2 参数 3…

其中，命令名为某个可执行的应用程序名（如：.exe 结尾的文件），各参数之间用空格分隔。

下面是对可执行文件 CSHARP5_12.exe 的命令行调用：

```
CSHARP5_12 one two three
```

运行及显示结果如图 5-9 所示。

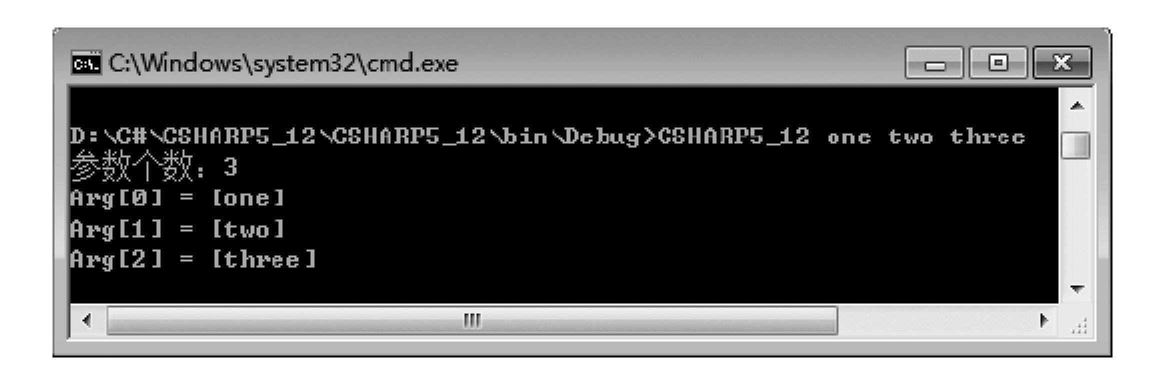


图 5-9　使用命令行实参

说明：如果在 Visual Studio 环境中运行应用程序，可以在“项目设计器”→“调试”页中指定命令行参数。具体操作方法：打开项目菜单，打开属性选项卡，选择调试，在命令行参数的文本框中输入参数，参数间用空格分隔。如输入三个参数：one two three，如图 5-10 所示。

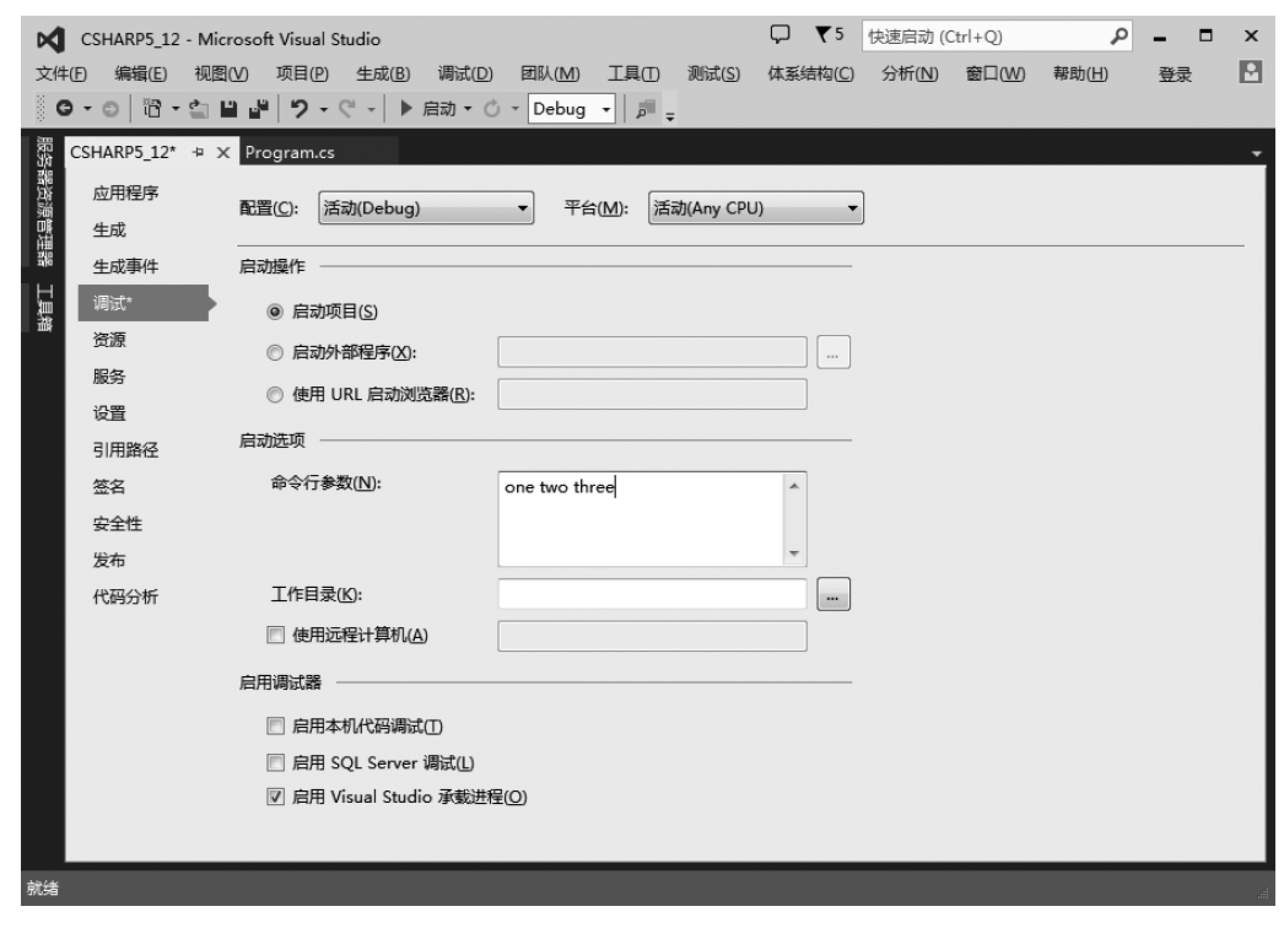


图 5-10　输入参数

执行程序，运行结果如图 5-11 所示。

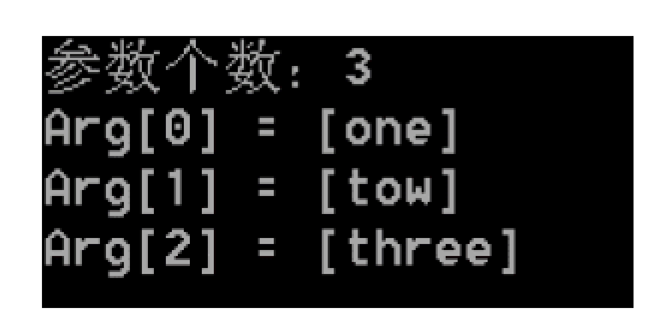


图 5-11　运行结果

习题

1. 对于数列：5,8,4,2,1,23,344,12,87，编写程序完成以下功能：

(1) 输出数列，每个数据占 5 位宽度。

(2) 求数列的和值及平均值(保留 2 位小数)。

(3) 找数：任意输入一个数，如果该数出现在数列中则输出“该数找到，其下标为 XX。”；否则输出“该数未找到！”。

2. 在 99 个元素的整型数组 arr 中找出最大值和最小值，并输出最大值和最小值以及对应的下标值(数组值随机产生)。

3. 利用一维数组求解问题。一家公司以底薪加提成的方式付给销售人员工资。销售人员每周获得 200 美元的底薪，外加本周达到一定销售额的 9%的提成。例如，一个销售人员一周的销售额是 5000 美元，就会得到 200 美元加上 5000 美元的 9%，即总共 650 美元。编写一个程序(利用一个计数器数组)，判断有多少销售人员可以获得以下范围的报酬(假设每个销售人员的报酬都将取整)。

(1) $200～$299；

(2) $300～$399；

(3) $400～$499；

(4) $500～$599；

(5) $600～$699；

(6) $700～$799；

(7) $800～$899；

(8) $900～$999；

(9) $1000 及以上。

4. 编写一个程序，模拟掷两个骰子。程序应该先随机掷出第一个骰子，再随机掷出第二个骰子。然后计算两个值的和(注意：由于每个骰子显示 1～6 之间的一个整数值，因此这两个值的和在 2～12 之间变动，其中 7 是出现频率最高的值，2～12 是出现频率最低的值)。图 5-12 显示这两个骰子值 36 种可能的组合。程序应该掷出这两个骰子 3600 次。请利用一个一维数组记录每个可能的和出现的次数。以表格的形式打印结果，并判定两个值的和是否合理(例如，有 6 种方式可以掷到 7，因此，所有掷出的和值，大约有 1/6

是7)。

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

图5-12　习题4图

5. 利用一维数组求解问题。读入20个数,每个数在10～100之间,包括10和100。在读入每个数时,确认这个数的有效性,并且若它和之前读入的数不一样,就把它存储到数组中。读完所有数之后,仅显示用户输入的不同的数值。

6. 利用二维数组求解问题。一家公司有4名售货员(编号为1～4),卖5种不同的产品(编号为1～5)。每天对于每种不同产品的销售情况,每位售货员都要递交一张相应的纸条。每张纸条包含以下内容:

- 售货员编号;
- 产品编号;
- 该产品当天的销售总额。

因此,每位售货员每天会上交0～5张销售纸条。假设现在有上个月所有纸条的信息。请编写一个程序,读入上个月的销售信息,统计每位售货员每种产品的销售总额。所有的总额应存储在一个二维数组sales中。处理完上个月的信息后,以表格形式打印出结果,每列表示一位售货员,每行表示一种产品。统计每行求出的上个月每种产品的销售总额;统计每列求出的上个月每位售货员的销售总额。打印输出的表格应该在相应行的右边和相应列的下面显示这些统计结果。

第6章 方　法

方法[①]是类中重要的成员。方法用于实现由对象或类执行的计算或操作。类中的方法是包含一系列语句的代码块。在C#程序设计中,方法几乎无处不在。如在前面各章的例题中,多次用到.NET Framework类库中Console类的WriteLine方法、ReadLine方法。熟练使用类库中定义的方法将会使编写程序更加快捷、迅速。除此之外,我们还必须掌握用户自定义类以及方法的编写过程。本章将介绍C#中方法的声明与使用方法。

6.1 C#的代码包装

在C#中,代码包装通常采用三种途径实现,分别是方法、类和名字空间。

使用方法可将程序按模块化设计。其目的一是"分治",可以利用小的、简单的功能块构造程序,使程序开发更易管理。二是实现软件的复用,即利用现成的方法作为"积木块",构建新程序。第三个目的是避免重复代码。

使用类可以组合用户自定义的类、方法、属性和.NET框架类库(FCL)以及其他类库中预定义的类、方法与属性。其中FCL标准类库提供丰富的函数,可以进行常用数据计算、字符串操作、字符操作、输入/输出操作,错误检查、Web程序开发等。

在C#中,将多个相关类组合成名字空间。FLC中每个类都属于一个名字空间。如Console类对应的名字空间是System。

6.2 静态方法和静态变量

在类中,使用static修饰符声明的成员称为静态成员。静态成员属于类所有,不需预先创建类的实例就能访问。静态成员的访问形式为"类名.静态成员",而不是"对象名.静态成员"。即静态成员是直接从类级访问,而不是从实例级访问(这里C#与C++不同)。

使用static声明的成员(域、字段)变量称为静态变量。如果类的所有对象要共享某个成员变量,那么可将该变量定义为类的静态变量。与实例成员变量不同,类的静态变量仅占有一块内存空间,且该空间是不能被撤销的。

---

① 方法在C#中又称为类的成员函数。

使用 static 声明的方法称为静态方法。静态方法是一种特殊的方法。在静态方法中只允许访问类的静态成员，而不允许访问其实例成员（即非静态成员）；但在实例方法（即非静态方法）中，可以访问类中的任何成员（包括静态和非静态成员）。

例 6-1 静态方法和静态变量（分析成员访问的正确与错误）。

程序代码如下：

```
using System;
namespace CSHARP6_1
{
    class Sample
    {
        public int x;                          //非静态变量
        public static int y;                   //静态变量
        public void f()                        //非静态方法
        {
         x=2;                                  //正确,等价于 this.x=2;
         y=2;                                  //正确,等价于 Sample.y=2;
        }
        public static void g()                 //静态方法
        {
            x=3;                               //错误
            y=3;                               //正确,等价于 Sample.y=3;
        }
    }
    class Program
    {
        static void Main(string[] args)        //静态方法
        {
        Sample s=new Sample();                 //创建实例 s
        s.x=1;                                 //s 访问公有的非静态成员 x-正确
        s.y=1;                                 //错误
        Sample.x=1;                            //错误
        Sample.y=1;                            //正确
        s.f();                                 //正确
        s.g();                                 //错误
        Sample.f();                            //错误
        Sample.g();                            //正确
        Console.WriteLine("{0},{1}",s.x,Sample.y);
        Sample s1=new Sample();                //创建实例 s1
        Console.WriteLine("{0},{1}",s1.x,Sample.y);
        }
    }
}
```

代码分析：

08～12 行为非静态方法 f()，其中：

第 10 行访问非静态成员 x 正确，该行等价于

```
this.x=2;
```

第 11 行访问静态成员 y 正确，该行等价于

```
Sample.y=2;
```

13～17 行为静态方法 g()，其中：

第 15 行 x=3；错误，静态方法不能访问非静态变量 x。

第 16 行 y=3；正确，静态方法能访问静态变量 y，该语句等价于

```
Sample.y=3;
```

21～35 行为静态方法 Main()，其中：

23 行：Sample s = new Sample()；创建 Sample 的一个实例 s。

25 行：s.y = 1；错误，不能使用实例 s 访问静态变量 y，可使用

```
Sample.y=1;
```

26 行：Sample.x = 1；错误，不能用类名 Sample 访问非静态成员 x。

27 行：正确，通过类名 Sample 访问公有的静态变量 y。

28 行：正确。将 s.x 改为 2，将静态成员 y 的值改为 2。

29 行：s.g()；错误，不能通过实例 s 访问静态方法 g()。

30 行：Sample.f()；错误，不能用类名 Sample 访问非静态方法 f()。

31 行：Sample.g()；正确，使用类名 Sample 访问该类的静态方法 g()。将静态成员 y 的值修改为 3。

32 行：Console.WriteLine("{0}，{1}", s.x, Sample.y)；正确，将输出 2 和 3。

33 行：Sample s1 = new Sample()；创建另一个实例 s1。这时，s1.x 默认初值为 0。

34 行：Console.WriteLine("{0}，{1}", s1.x, Sample.y)；正确，将输出 0 和 3。

6.3 关于方法声明与使用

方法在类或结构中声明，声明时需要指定访问级别（即访问修饰符）、返回值、方法名、方法参数以及方法体。方法参数应放在一对圆括号中，空括号表示方法无须参数。

1. 方法的声明

方法声明的一般格式：

```
访问修饰符  返回值类型  方法名(方法参数)
{
    语句序列
```

```
}
```

其中：

- 方法的访问修饰符常用 public，以保证在类定义外部能够调用该方法。
- 方法的返回值类型用于指定由该方法计算并返回的值的类型，可以是任何值类型或引用类型。如果方法不返回一个值，则返回值类型为 void。
- 方法名是合法的 C# 标识符。
- 方法参数应放在一对圆括号中，方法参数必须指定参数的类型和参数名称。如果有多个参数，各参数之间需用逗号分隔。
- 大括号({})括起来的语句序列称为方法体，实现方法特定的功能。如果方法有返回值，则方法体中必须包含一个 return 语句，以指定返回值，其类型与方法的返回值类型相同。如果方法无返回值，则在方法体中可以不包含 return 语句，或包含一个不指定任何值的 return 语句。

例如，代码段

```
public int Max(int x,int y)
{
    return x>y?x:y;
}
```

声明一个公有的、带 2 个整型参数 x 和 y 以及返回整型值的方法 Max，其功能是找出 x 和 y 中的大数并返回其最大数。

例如，代码段

```
public static void SayHello()
{
    Console.WriteLine("Hello,World!");
}
```

声明一个公有的、无参数、无返回值的静态方法 SayHello，其功能是在屏幕显示字符串“Hello，World!”。

2. 方法的使用

方法的使用通过对方法的调用来实现。按照方法有无返回值，采用如下调用形式。

(1) 无返回值：方法调用以独立语句的形式出现。

```
对象名.方法名(实参列表);        //非静态方法调用
```

或

```
类名.方法名(实参列表);          //静态方法调用
```

例如，无返回值的 Console 类的静态方法 WriteLine 的调用格式：

```
Console.WriteLine(…);
```

(2) 有返回值：方法调用仅作为表达式的形式出现。

例如，调用 Math 类的静态方法 Sqrt。

```
y=Math.Sqrt(x);                    //求 x 的平方根
```

注意：调用带参数的方法时，方法名后的圆括号内应提供实参。如果有多个实参，实参之间需用逗号分隔。通常情况下，调用方法时实参的个数及类型应与方法声明中形参的个数、类型对应一致。

例 6-2　阅读程序代码，分析完成的功能。

程序代码如下：

```
using System;
//找最大数
namespace CSHARP6_2
{
    class MaxNum
    {
     public int Max(int x,int y)
     {
         return x > y ? x : y;
     }
     public static void Main(string[] args)
     {
         MaxNum a= new MaxNum();              //创建对象 a
         int z;
         z= a.Max(25,30);                     //通过对象 a 调用方法 Max
         z= a.Max(34,z);                      //通过对象 a 再次调用方法 Max
         Console.WriteLine("Max= {0}",z);     //输出 z
     }
    }
}
```

程序运行结果：

```
Max= 34
```

问题：

(1) 程序完成的功能是什么？

(2) 如果将 Max 声明为静态方法，如：

```
public static int Max(int x,int y)
{
    return x > y ? x : y;
}
```

那么主调方法应如何修改？

6.3.1 方法参数修饰符

方法中经常涉及“方法参数”。一般情况下，主调方法通过方法参数向被调方法传递数据，被调方法会返回一个结果值供主调方法使用。除此之外，如需在被调方法中更改实参的值，或向主调方法传回更多的数据值，那么可以借助方法参数修饰符(如 ref、out)来完成。参数修饰符用于控制参数的传递方式，不同的修饰符具有不同的传递方式。C# 中方法参数共有四种修饰，分别为无任何修饰符、ref、out 和 params[①]。下面分别介绍四种参数的不同传递方式。

1. 值参数

值参数即不使用任何修饰符标记的参数。值参数采用值方式传递(简称值传递或默认传递)。值方式传递是指在进行方法调用时，系统将实参的值复制给形参变量(即形参是实参值的一个副本)，这时，方法中可使用或修改形参变量的值，但不会影响或改变实参的值。因此，值方式传递是一种安全的传递方式。

例 6-3 值参数的应用。

程序代码如下：

```
01: using System;
02: namespace CSHARP6_3
03: {
04:     class Program
05:     {
06:         public static void Swap(int x, int y)      //形参为值参数
07:         {
08:             int temp = x;
09:             x= y;
10:             y= temp;
11:         }
12:         public static void Main(string[] args)
13:         {
14:             int i = 1, j = 2;
15:             Console.WriteLine("i= {0}, j= {1}", i, j);
16:             Swap(i,j);                             //实参为值参数
17:             Console.WriteLine("i= {0}, j= {1}", i, j);
18:         }
19:     }
20: }
```

程序分析：

第 6 行方法定义中的形参属于“值参数”(无任何参数修饰符修饰)，因此第 16 行的方

① params 修饰符已在第 5 章介绍。

法参数将采用“值传递”。调用时，系统将实参 i 和 j 的值传递给对应的形参变量 x 和 y，方法中交换 x 和 y 的值，而实参 i 和 j 的值将保持不变。这就是“值传递”。

程序执行结果：

```
i=1,j=2
i=1,j=2
```

由此可以看出，采用“值传递”，形参获取的是实参值的“副本”，方法中只对“副本”操作，而对应的原值不变。那么，如何通过交换 Swap 中的形参 x 和 y，就能使实参 i 和 j 的值交换呢？我们需引入参数的另一种传递方式，即“引用传递”。

2. 引用参数（使用 ref 修饰符标记的参数）

引用参数采用引用传递。引用传递是指实参传递给形参时，不是将实参变量的值复制给形参，而是将实参的引用（即实参变量的地址）复制给形参，这时，形参作为实参的别名与实参共享同一内存单元。因此，方法中使用形参就如同使用实参一样，形参的值变了，实参的值也就变了。

例 6-4 引用参数的应用（改写例 6-3）。

程序代码如下：

```
01: using System;
02: namespace CSHARP6_4
03: {
04:     class Program
05:     {
06:         public static void Swap(ref int x, ref int y)     //形参为引用参数
07:         {
08:             int temp=x;
09:             x=y;
10:             y=temp;
11:         }
12:         public static void Main(string[] args)
13:         {
14:             int i=1, j=2;
15:             Console.WriteLine("i={0}, j={1}", i, j);
16:             Swap(ref i,ref j);                            //实参为引用参数
17:             Console.WriteLine("i={0}, j={1}", i, j);
18:         }
19:     }
20: }
```

程序分析：

第 6 行方法中的 2 个形参都是“引用参数”（由 ref 修饰），因此，执行第 16 行的方法调用时，其中 2 个参数将采用“引用方式传递”。即调用时，系统将实参 i 和 j 的“引用”（即 i

和 j 的"地址")传递给对应的形参变量 x 和 y,这时,使用 x 可以引用实参 i,使用 y 可以引用实参 j。因此,方法中看似交换 x 和 y,实际上在交换实参变量 i 和 j。这就是"引用传递"。

程序执行结果:

```
i=1,j=2
i=2,j=1
```

可以看出,采用"引用传递",可以通过形参来改变实参的值。

特别提醒:

(1) 对于引用参数,无论是方法声明还是方法调用,必须使用 ref 修饰符显式标记参数。

(2) 用作实参的引用参数必须显式初始化。

3. 输出参数(使用 out 修饰符标记的参数)

输出参数与引用参数类似,也是采用"引用传递"。在 C# 中,通过设置输出参数,可从方法中传回更多的数据值。需注意的是,输出参数在方法调用前不必显式初始化,这点与引用参数不同。

特别提醒:

(1) 对于输出参数,无论是方法声明还是方法调用,必须使用 out 修饰符显式标记参数。

(2) 输出参数在方法调用前不必显式初始化,但在方法中必须为该参数提供数据值(作为该参数的返回值)。

例 6-5 输出参数的应用。

有一个算术运算方法 Cal(),使用输出参数,返回两个整数的和、差、积和商。

程序代码如下:

```
using System;
namespace CSHARP6_5
{
    class Program
    {
        //方法 Cal()中可设置四个输出参数
        static void Cal(int x,int y,out int add,out int sub,out int mul,out int div)
        {
            add=x+y;
            sub=x-y;
            mul=x*y;
            div=x/y;
        }
        //调用 Cal()的主方法
        static void Main(string[] args)
        {
            int a=9,b=3;
```

```
18:             int sum,sub,mul,div;
19:             Cal(a,b,out sum,out sub,out mul,out div);
20:             Console.WriteLine("{0}+{1}={2}",a,b,sum);
21:             Console.WriteLine("{0}-{1}={2}",a,b,sub);
22:             Console.WriteLine("{0}*{1}={2}",a,b,mul);
23:             Console.WriteLine("{0}/{1}={2}",a,b,div);
24:             Console.ReadKey();
25:         }
26:     }
27: }
```

程序说明：

07 行说明方法 Cal 带有 6 个参数，其中有 2 个值参数（无参数修饰符），用于接收实参的值，4 个输出参数（由 out 修饰），将通过引用 4 个实参输出参数带回方法中的 4 个计算结果。09～12 行分别为 4 个输出参数提供结果数据，即 x 与 y 的加、减、乘、除运算的结果。在主方法中，第 17 行定义变量 a 和 b，作为方法调用时的实参（为值参数），并已初始化。第 18 行定义 4 个变量 sum，sub，mul 和 div，用作方法调用时的输出参数，因此，该变量不必初始化。第 19 行调用方法 Cal。20～23 行分别输出加、减、乘、除运算的结果。在这里，2 个值参数采用“值传递”，而 4 个输出参数采用“引用传递”。

程序输出结果：

```
9+3=12
9-3=6
9*3=27
9/3=3
```

6.3.2　参数传递的隐式转换与强制转换

在方法调用中，若实参类型与形参类型不同，默认情况下系统会依照“类型转换规则”隐式地将实参转换成形参类型传递给形参。例如：Math 类的方法 Sqrt，参数声明为 double 型，但可以使用整型参数调用。下列语句：

```
Console.WriteLine(Math.Sqrt(9));
```

能正确求值 Math.Sqrt(9)，输出值为 3.0。方法声明的参数表使 C＃将 int 值 9 转换成 9.0，然后传入 Sqrt 方法。想进行这类转换时，如果不满足 C＃类型转换规则，则会产生编译错误。隐式转换规则指定允许哪些转换，即进行哪些转换不会丢失数据。在上述 Sqrt 例子中，int 转换成 double 不会丢失数据。但将 double 转换成 int 会使 double 的小数部分截尾，从而使部分数值丢失。另外，double 可以存放比 int 变量更大（和更小）的值，因此，将 double 赋予 int 可能因 double 值无法放进 int 中而丢失。将较大的整数类型转换为较小的整数类型（如 long 变成 int）也可能改变数据值。

类型转换规则适合于包含两个或多个简单类型构成的表达式或作为方法参数传递的简单类型值。每个值转换为表达式中的适当类型。表 6-1 按字母顺序列出简单类型可转

换的类型。注意,在C#中所有简单类型都可以隐式转换为object类型。

默认情况下,如果形参类型不能表示实参类型,则C#不允许它们之间进行隐式转换(如int值2000000就不能用short表示,任何有小数的浮点数不能用long,int或short之类的整数表示)。因此,为了防止编译错误,避免简单类型隐式转换造成信息丢失,编译器要求用类型转换运算符显式强制转换,这样就可以“控制”编译器。假设生成Square方法,计算整数的平方,要求int参数;调用Square时,如果使用double参数doubleValue,则要将方法调用写成Square((int)doubleValue)。这个方法调用显式地将doubleValue值转换成int后,在Square中使用。这样,如果doubleValue值为4.5,则方法取4,返回16,而不是20.25(造成信息丢失)。

表 6-1 简单类型的隐式转换规则

类　型	隐式转换类型
bool	不能隐式转换成其他简单类型
byte	ushort,short,uint,int,ulong,long,decimal,float 或 double
char	ushort,uint,int
decimal	不能隐式转换成其他简单类型
double	不能隐式转换成其他简单类型
float	double
int	long,decimal,float 或 double
long	decimal,float 或 double
sbyte	short,int,long,decimal,float 或 double
short	int,long,decimal,float 或 double
uint	ulong,long,decimal,float 或 double
ulong	decimal,float 或 double
ushort	uint,int,ulong,long,decimal,float 或 double

6.3.3 方法重载

方法重载是面向对象对结构化编程特性的一个重要扩充,利用方法重载完成类似的任务可以使程序易于阅读和理解。方法重载允许类中两个或两个以上的方法(包括隐藏继承而来的方法)取相同的名字,这些方法可以完成不同的功能。编译器根据实参和形参的个数、类型和顺序合理选择一个方法进行调用。构成重载的方法具有以下特点:

(1) 方法名相同。

(2) 方法参数列表不同,满足如下三点之一即可。

- 方法参数数目不同。
- 方法拥有相同数目的参数,但参数的类型不一样。
- 方法拥有相同数目的参数和参数类型,但是参数类型出现的先后顺序不一样。

特别提醒：构成重载的方法与方法的返回值类型无关。

例 6-6 重载方法 cube 计算 decimal 类型值的立方以及 double 类型值的立方。

程序代码如下：

```
01: //重载方法 cube 计算 decimal 类型值的立方以及 double 类型值的立方
02: using System;
03: namespace CSHARP6_6
04: {
05:     class program
06:     {
07:         public static decimal cube(decimal x)
08:         {
09:             return x * x * x;
10:         }
11:         public static double cube(double x)
12:         {
13:             return x * x * x;
14:         }
15:         public static void Main()
16:         {
17:             Console.WriteLine("decimal 数 2.0m 的立方是: {0}",cube(2.0m));
18:             Console.WriteLine("双精度数 2.5 的立方是: {0}",cube(2.5));
19:             Console.ReadKey();
20:         }
21:     }
22: }
```

程序说明：

07～10 行和 11～14 行为 cube 重载方法声明。第 17 行中 cube(2.0m)将调用参数类型为 decimal 的 cube 方法(2.0m 为 decimal 常数)；而第 18 行中 cube(2.5)将调用参数类型为 double 的 cube 方法(2.5 为 double 常数)。

程序运行结果：

```
decimal 数 2.0m 的立方是: 8.000
双精度数 2.5 的立方是: 15.625
```

6.3.4 可选参数和命名参数

可选参数是指在声明方法时给方法的某些特定参数(即特定形参)指定默认初值，这样在调用方法时可省略这些参数对应的实参。

例如：带有可选参数的方法。

```
public static int Days(int day,int month,int year=2015)
{
    …
}
```

其中：形参 year 为可选参数，默认初值是 2015。

所以，以下调用形式均合法。

形式一：

```
Days(18,2)
```

其中，18 传递给形参 day，2 传递给形参 month，而形参 year 将使用其默认初值 2015。

形式二：

```
Days(18,2,2016)
```

其中，18 传递给形参 day，2 传递给形参 month，2016 传递给形参 year(原默认初值失效)。

特别提醒：

(1) 可选参数必须位于所有必选参数之后，即可选参数出现在参数列表的右端。

(2) 如果调用时为多个可选形参中的任意一个提供实参，则必须为前面的所有可选形参提供实参。

例如：下面的方法调用将导致编译错误。

方法声明如下：

```
public static double fun(double a,double b=20.0,double c=15.0)
{
    …
}
```

方法参数中 a 为必选参数，b 和 c 为可选参数。

方法调用：

```
Console.WriteLine(fun(5.0,,12.3));
```

存在编译错误，其原因是：fun 方法调用中为第 3 个可选参数提供了实参值 12.3，而它前面的第 2 个可选参数没有提供实参。

若将上述调用修改为：

```
Console.WriteLine(fun(5.0,c:12.3));
```

则是正确的选择。其中方法调用 fun 的第 2 个实参形式为“c:12.3”称为命名参数，意为将 12.3 传递给形参 c。

命名参数是指在方法调用时，使用形参名为实参命名，其形式为：

```
形参名:实参
```

意为实参将传递给指定的形参。使用命名参数可实现参数之间按名传递，这样参数结合时与参数位置次序无关。

例如：使用命名参数调用 Days 方法。

调用一：

```
Days(month:2,day:18)
```

其中,命名参数 month:2 意为将 2 传递给形参 month;命名参数 day:18 意为将 18 传递给形参 day;而形参 year 使用默认初值 2015。

调用二:

```
Days(year:2016,month:2,day:18)
```

其中:命名参数 year:2015 意为将 2016 传递给参数 year。命名参数 month:2 意为将 2 传递给形参 month;命名参数 day:18 意为将 18 传递给形参 day;

例 6-7　可选参数和命名参数。

程序代码如下:

```
01: //可选参数和命名参数
02: using System;
03: namespace CSHARP6_7
04: {
05:     class Program
06      {
07:         //方法 Add带有可选参数
08:         static int Add(int a,int b=2)
09:         {
10:             return a+b;
11:         }
12:         static void Main(string[] args)
13:         {
14:             Console.WriteLine(Add(1));
15:             Console.WriteLine(Add(1,3));
16:             //使用命名参数调用 Add,实参顺序和形参顺序可以不同
17:             Console.WriteLine(Add(b: 6,a: 1));
18:             Console.ReadKey();
19:         }
20:     }
21: }
```

方法 Add(08～11 行)中形参 a 为必选参数,而 b 为可选参数,默认值为 2。第 14 行 Add(1)是将 1 传递给形参变量 a,b 将使用默认值 2。第 17 行 Add(b: 6, a: 1) 使用命名参数,很明显,将 6 传递给 b,将 1 传递给 a。

程序运行结果:

```
3
4
7
```

6.3.5　按值传递与按引用传递

我们知道,C# 中的数据类型分为值类型和引用类型两大类。值类型直接存储数据

值,其值保存在内存的栈(stack)空间中。引用类型存储的是对数据值的引用,即存储数据所在的存储单元地址。引用类型涉及数据和存储数据的内存单元地址两个概念,在C#中系统使用栈(stack)和堆(heap)协同处理引用类型数据。并约定,堆空间保存实际数据(值),而栈空间保存数据的内存单元地址。所以,访问引用类型数据时,总是先从栈中找到“数据存放的地址”,再通过“地址”存取堆中的数据。在实际编程中,用户只是通过分配在栈中的变量存取数据。对于引用类型变量,系统会根据变量的当前值间接访问堆中的数据。

C#预定义的简单类型,int、float、double、bool 和 char 等都是值类型;另外,enum(枚举)、struct(结构)也是值类型;string(字符串)、数组、类等都是引用类型。

下面举例说明值类型和引用类型的内存分配。

例如,有如下定义:

```
int a=8;
int[] b=new int[5]{80,98,65,78,90};
```

其中:

a 是整型变量,属于值类型。a 的值分配在栈中,程序中直接使用 a 变量访问。

b 是整型数组,属于引用类型,因此 b 是一个引用整型数组的变量。该数组值存放在堆中一块连续的存储单元中,而 b 变量分配在栈中,其值为存放在堆中数组的首地址。程序中使用栈中的 b 变量间接访问堆中数组的元素。其内存分配示意如图 6-1 所示。

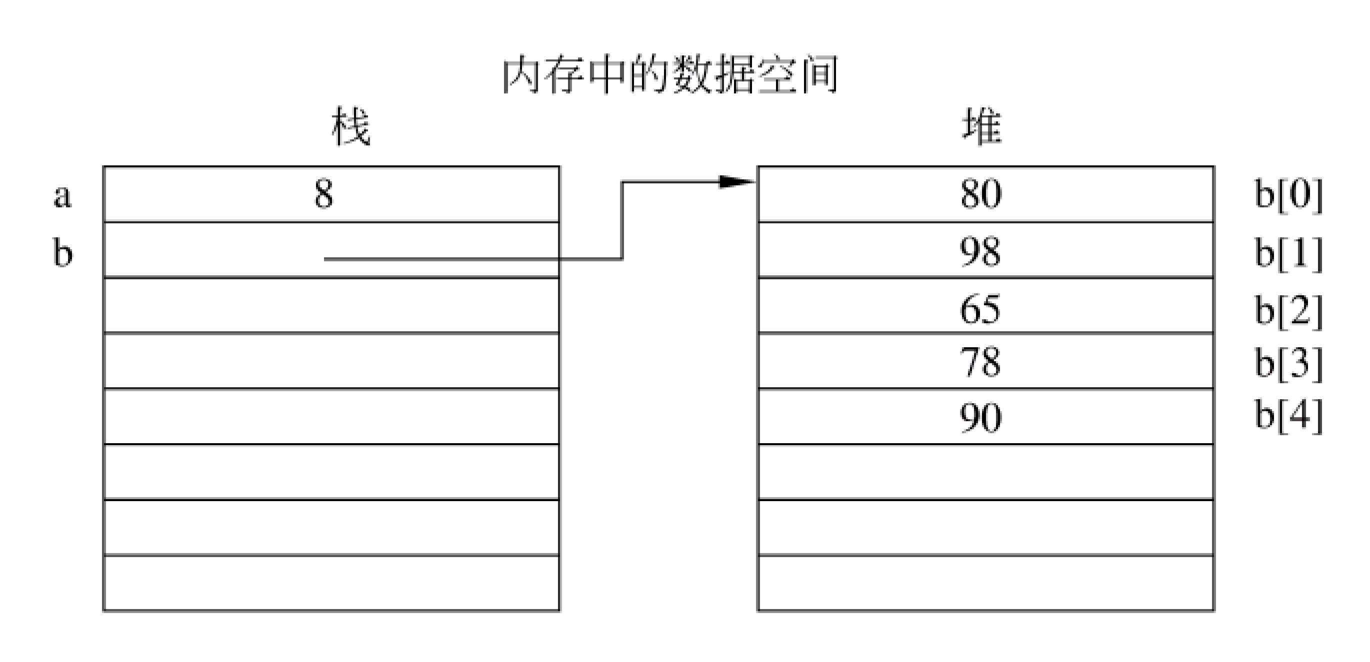


图 6-1　变量与数组的内存分配

在 C#中,参数的传递方式分为按值传递和按引用传递两种。规定:不使用任何参数修饰符标记的方法参数将采用“按值传递”,这是系统默认的传递方式。若使用 ref 或 out 修饰符标记方法参数,则采用“按引用传递”。

特别提醒: 值类型参数不都是按值方式传递,引用类型参数也并非都是按引用方式传递。如:在方法 void Test1(int a)中,参数 a 属于值类型,而在方法 void Test2(int [] array)中,参数 array 属于引用类型,但它们的传递方式是相同的,都是采用“按值传递”。

1. 按值传递

在调用方法时,将实参值的副本传递给方法参数(形参),这时,形参具有与实参等值的、相互独立的存储空间。

（1）当实参为值类型参数时，方法中对形参进行的任何操作，都不会影响和改变对应实参的值。

例 6-8　值类型作为方法参数。

程序代码如下：

```
01: //值类型作为方法参数
02: using System;
03: namespace CSHARP6_8
04: {
05:     class Program
06:     {
07:         public static void fun(int a)
08:         {
09:             int b=5;
10:             a++;
11:             b+=a;
12:             Console.WriteLine("方法中 a={0},b={1}",a,b);
13:         }
14:         public static void Main(string[] args)
15:         {
16:             int a=1;
17:             Console.WriteLine("调用前 a={0}",a);
18:             fun(a);
19:             Console.WriteLine("调用后 a={0}",a);
20:         }
21:     }
22: }
```

代码分析：

主方法（14～21 行）的第 18 行的实参 a 为值参数，而方法 fun（07～13 行）中第 7 行的形参 a 同样为值参数，实参 a 与形参 a 分别有各自的存储单元，形参 a 接收实参 a 的值，其初值均为 1。之后，执行方法中的 10～12 行，这时方法中 a 的值改变为 2，而主方法中 a 值保持不变，方法调用后 a 的值仍为 1。

程序执行结果如下：

```
调用前 a=1
方法中 a=2,b=7
调用后 a=1
```

（2）当实参为引用类型参数时，方法中使用形参可以间接访问实参所引用的堆空间中的数据，但方法中改变形参的值不会改变对应实参变量的值。

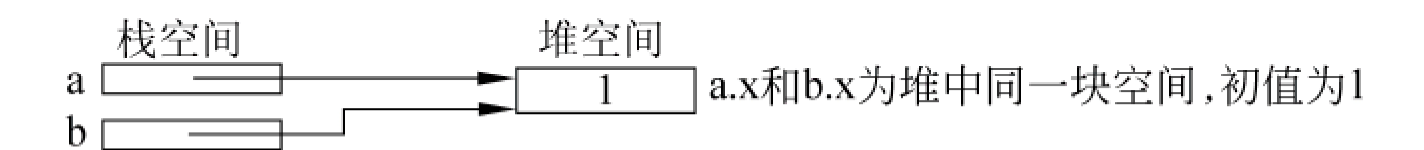
例 6-9　引用类型作为方法参数。

程序代码如下：

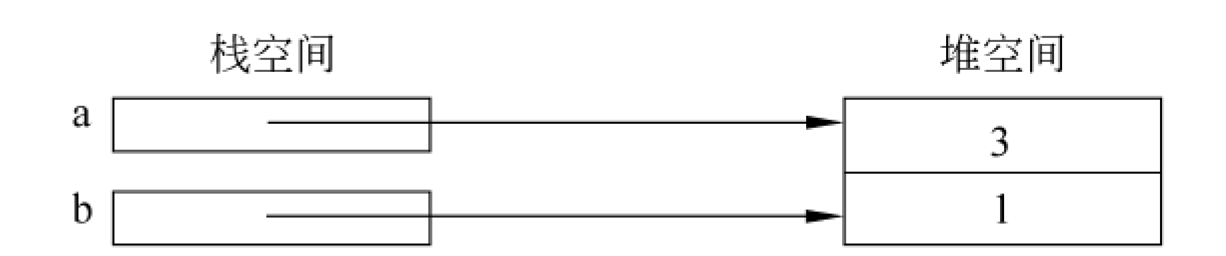
```
01: using System;
02: namespace CSHARP6_9
03: {
04:     class other
05:     {
06:         public int x=1;
07:     }
08:     class Program
09:     {
10:         static void fun(other b)             //引用类型变量作为形参
11:         {
12:             Console.WriteLine(" 在方法 fun中: ");
13:             Console.WriteLine(" b.x 的初值: {0}",b.x);
14:             b.x+=2;
15:             Console.WriteLine(" b.x 增 2后的值: {0}",b.x);
16:             b=new other();                   //更改形参 b的值
17:             b.x++;
18:             Console.WriteLine(" 更改形参 b,b.x增 1后的值为: {0}",b.x);
19:         }
20:         static void Main(string[] args)
21:         {
22:             other a=new other();             //创建对象 a
23:             Console.WriteLine("主方法调用 fun之前 a.x 的值: {0}",a.x);
24:             fun(a);                          //引用型变量作为实参,并按值传递
25:             Console.WriteLine("主方法调用 fun之后 a.x 的值: {0}",a.x);
26:         }
27:     }
28: }
```

程序分析:

04～07 行声明类 other。08～27 行声明主类 Program,包含主方法(20～26 行)和 fun 方法(10～19 行)。第 22 行创建 other 对象 a。24 行调用方法 fun,这时实参 a 为引用型变量,并"按值传递"将 a 的值传递给 b。这时内存分配如下:



执行方法中第 16 行后,b 的值发生变化,这时内存分配如下:



回到主方法后,内存状况如下:

栈空间 堆空间

a 3

程序执行结果:

主方法调用 fun之前,a.x 的值:1。
在方法 fun中,b.x的初值:1。b.x增 2后的值:3。更改形参 b,b.x增 1后的值为:2。
主方法调用 fun之后 a.x 的值:3。

例 6-10 采用“值传递”传递数组。

我们知道,数组可以作为方法参数进行传递。当“按值传递”数组时,向方法传递的是数组的引用(即数组在堆中的首地址);而方法可以通过接收该引用的形参间接使用或改变实参数组元素的值,但不可以通过改变形参变量的值来改变引用实参数组变量的值。

程序代码如下:

```
//采用"值传递",向方法中传递数组
using System;
namespace CSHARP6_10
{
    class Program
    {
        public static void valArr(int[] y)   //y是引用类型,但 y以值方式传递
        {
            y[0]=6;
            y[1]=7;
            y=new int[] { 60,70,80,90 };             //改变 y的值,使 y引用一个新创建的数组
        }
        public static void print(int[] x)    //x是引用类型,但 x以值方式传递
        {
            foreach(int t in x)
            Console.Write("{0} ",t);
            Console.WriteLine();
        }
        public static void Main()
        {
            int[] a=new int[] { 1,2,3,4,5 };
            print(a);
            valArr(a);
            print(a);
        }
    }
}
```

程序运行结果:

```
1 2 3 4 5
6 7 3 4 5
```

代码分析：

在主方法(19～25 行)中，21 行创建一维整型数组 a。其实，a 为引用数组的变量，其值为堆中分配的数组的首地址。数组 a 的内存分配状况如图 6-2 所示。

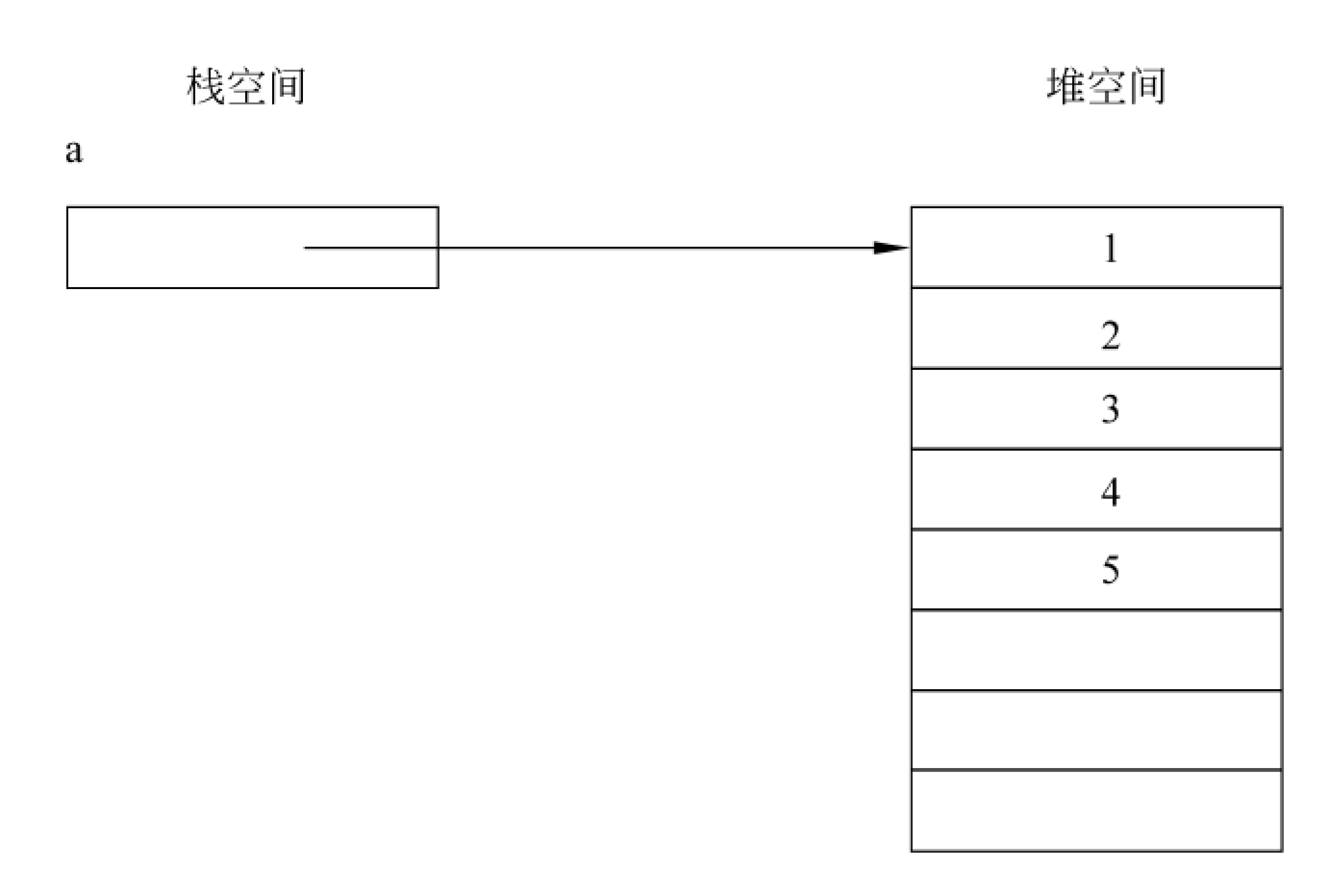


图 6-2　a 数组的内存状况

第 23 行调用方法 valArr。因方法参数无任何参数修饰符，所以实参 a 与形参 y 按“值传递”进行。这时，y 的值就是 a 的值，即 y 也指向堆中的数组，方法中可通过 y 访问堆中数组元素值。值传递后的内存分配状况如图 6-3 所示。

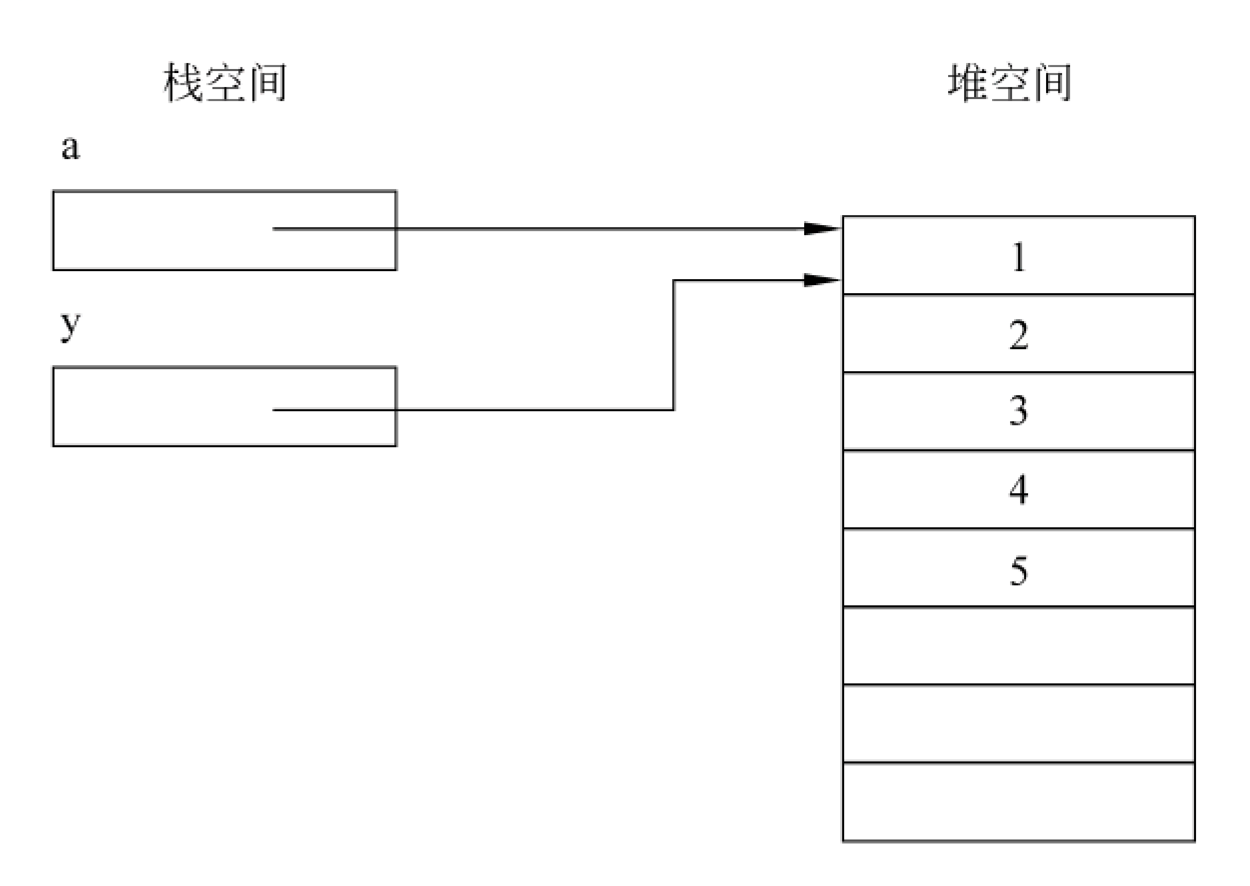


图 6-3　当前内存状况

在方法 valArr(07～12 行)中，第 09 行和第 10 行用于改变 y[0]和 y[1]的值，即改变堆中数组元素的值。这时的内存状况如图 6-4 所示。

第 11 行执行后，y 的值发生改变，即 y 不再指向堆中原数组，而指向新分配的数组。a 值保持不变，a 仍指向原数组。这时的内存分配状况如图 6-5 所示。

第 24 行输出 a 数组，注意 a 仍指向原数组，所以输出结果为：6 7 3 4 5。

结论：当按“值传递”传递引用类型数据时，方法中若改变形参的值(即更新作为形参

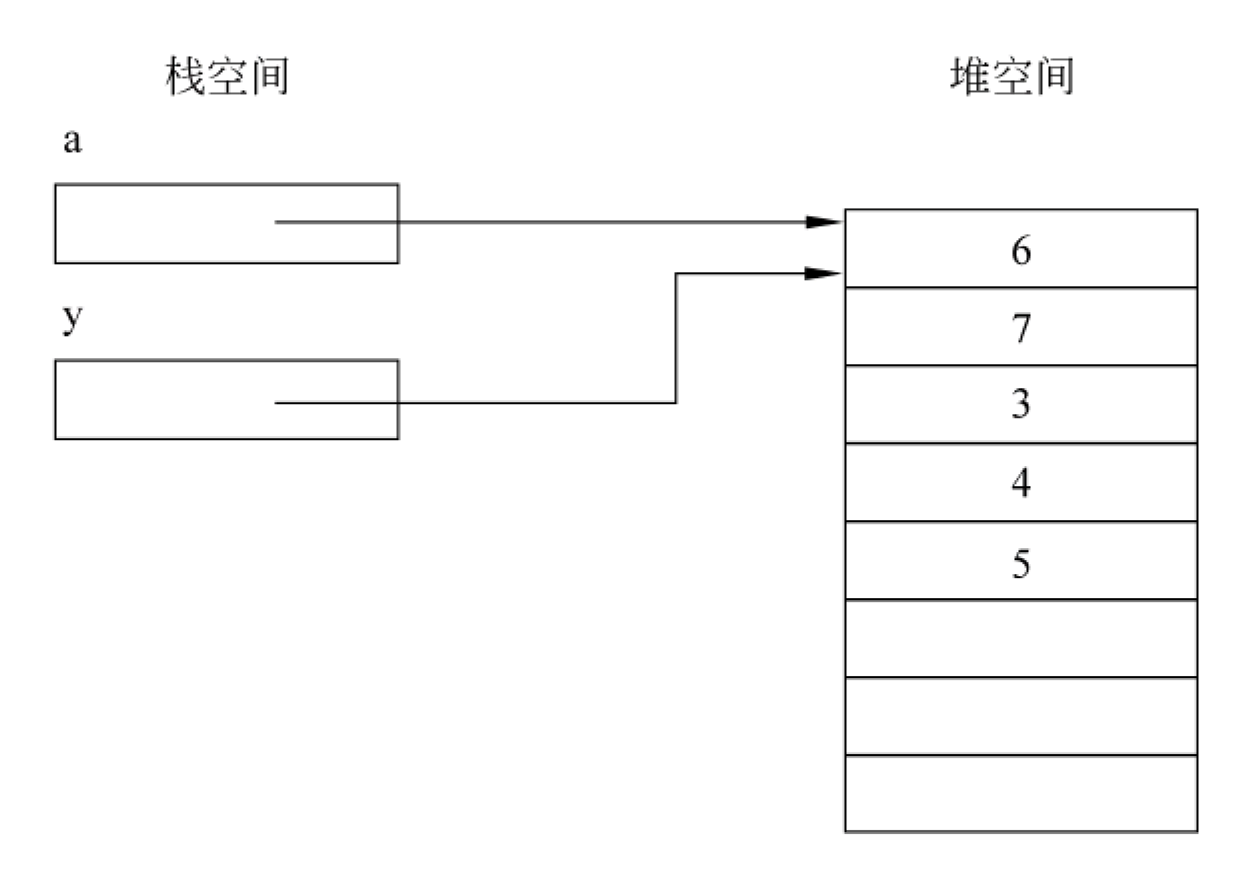


图 6-4 数组的变化(1)

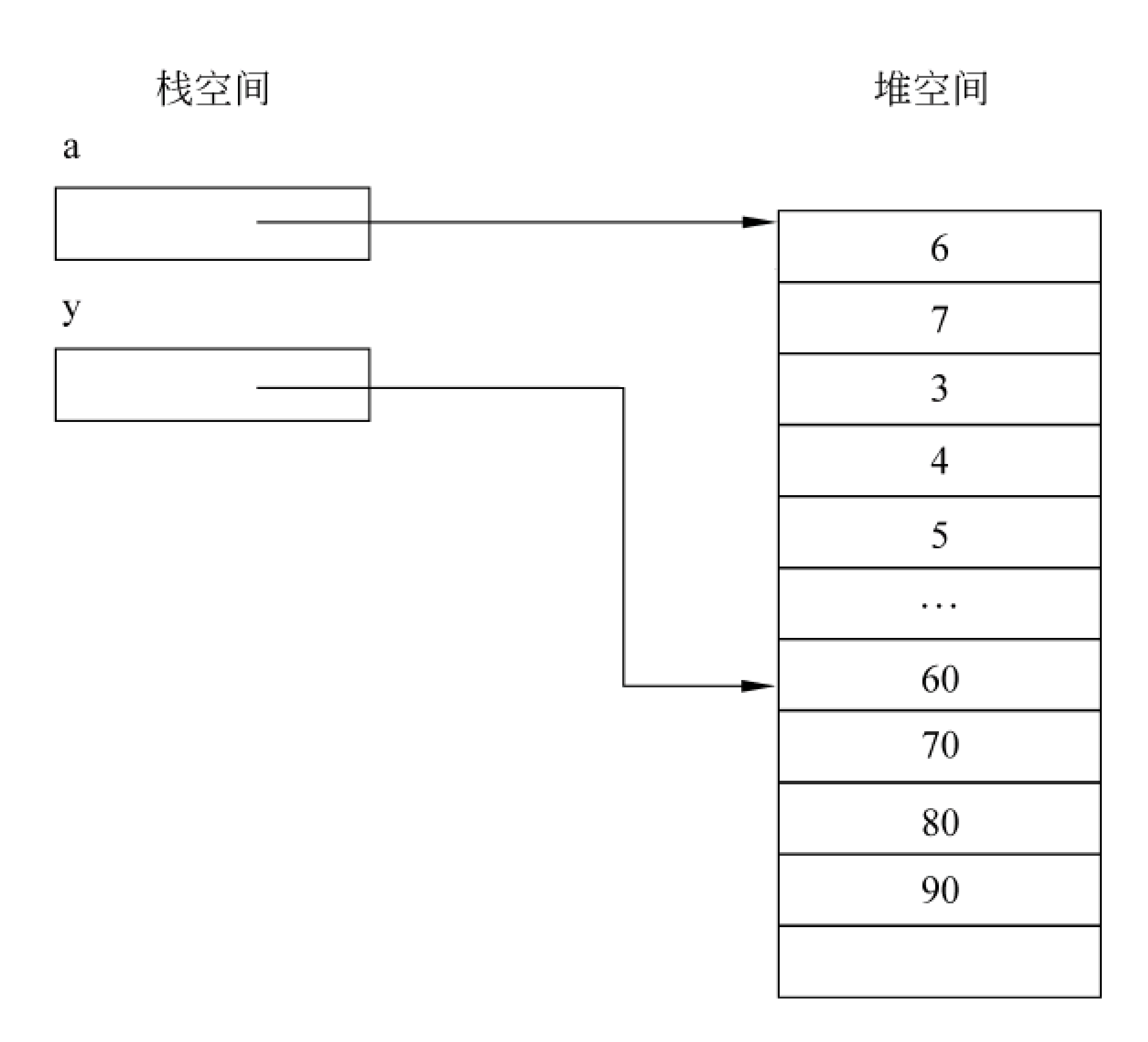


图 6-5 数组的变化(2)

的引用变量值),则对应实参的值保持不变。

2. 按引用传递(使用参数修饰符 ref 和 out)

在调用方法时,将实参的引用(即实参变量的地址)传递给形参。这时,方法通过形参可以间接访问实参,改变形参的值也就改变了实参的值。

例 6-11 修改例 6-10,按照“引用传递”向方法传递数组。

利用数组的“引用传递”,不仅可以通过形参改变实参数组元素的值,也可以改变引用实参数组变量的值。

程序代码如下:

```
//按引用传递方式向方法传递数组
using System;
```

```
03: namespace CSHARP6_11
04: {
05:     class Program
06:     {
07:         public static void valArr(ref int[] y)      //y为引用参数
08:         {
09:             y[0]=6;
10:             y[1]=7;
11:             y=new int[] { 60,70,80,90 };
12:         }
13:         public static void print(int[] x)           //x为值参数
14:         {
15:             foreach(int t in x)
16:             Console.Write("{0} ",t);
17:             Console.WriteLine();
18:         }
19:         public static void Main()
20:         {
21:             int[] a=new int[] { 1,2,3,4,5 };
22:             print(a);                               //a为值传递
23:             valArr(ref a);                          //a为引用传递
24:             print(a);                               //a为值传递
25:         }
26:     }
27: }
```

程序运行结果：

```
1 2 3 4 5
60 70 80 90
```

代码分析：

07～12 行为方法 valArr 的代码，其中第 7 行方法参数 y 为引用参数。13～18 行为方法 print 的代码，其中第 13 行方法参数 x 为值参数。19～25 行为主方法代码，其中第 21 行创建数组 a(又称 a 为引用数组的变量)。22 行输出 a 数组，结果为：1 2 3 4 5。23 行调用方法 valArr，因实参 a 为引用传递(用 ref 修饰)，它与值传递不同，这时，形参 y 接收实参 a 变量的地址。采用引用传递后，其内存分配状况如图 6-6 所示。方法 valArr 中通过 y 间接访问 a，通过 y[i]间接访问 a[i]。这时，在第 9 行中 y[0]与 a[0]等价，而在第 10 行中 y[1]与 a[1]等价。执行后，堆中的数组值变化如图 6-7 所示。第 11 行通过创建新的数组来改变 y 的值，其实也同时改变 a 的值，即 a 不再引用原数组。这时内存分配状况如图 6-8 所示。24 行执行后，继续输出 a 数组，结果为：60 70 80 90。

结论：当按“引用传递”传递引用类型数据时，方法中改变形参的值，同时也在改变实参的值。

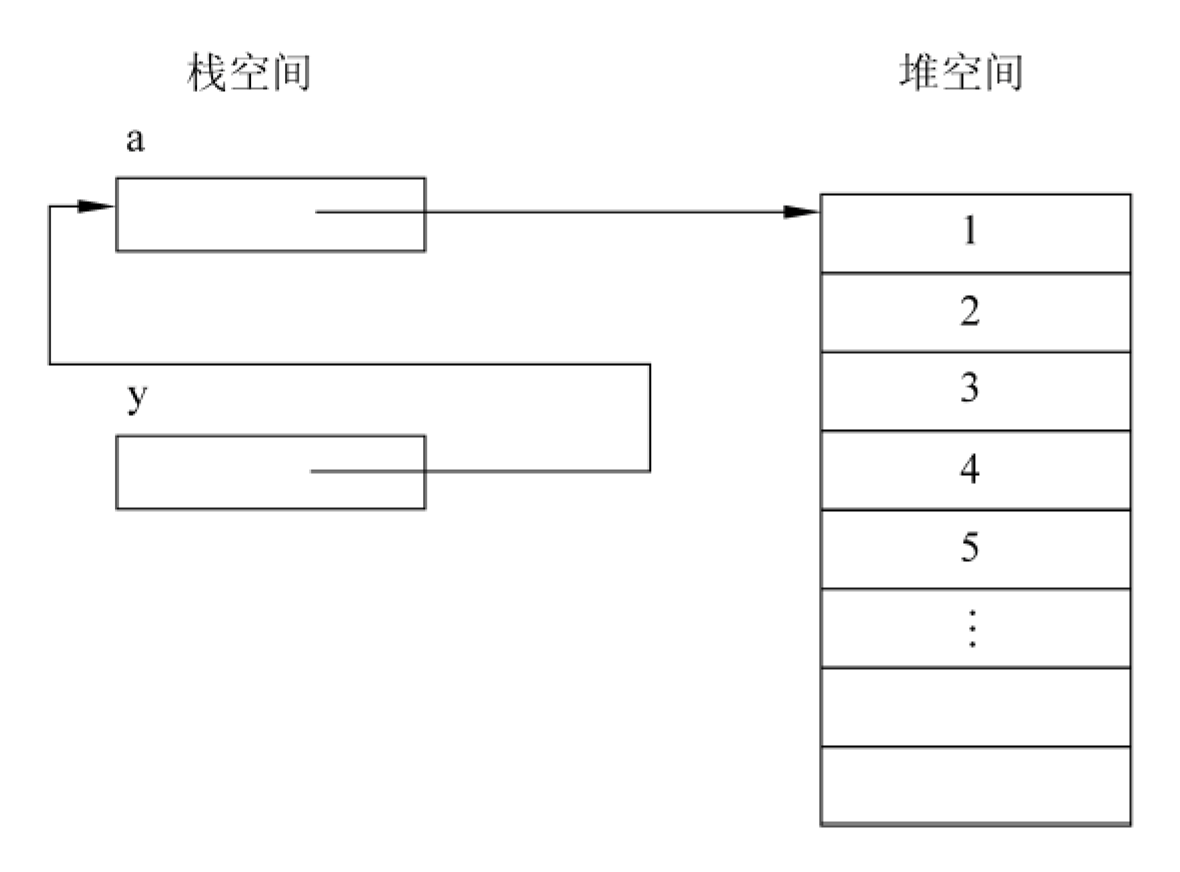


图 6-6　内存示意图(1)

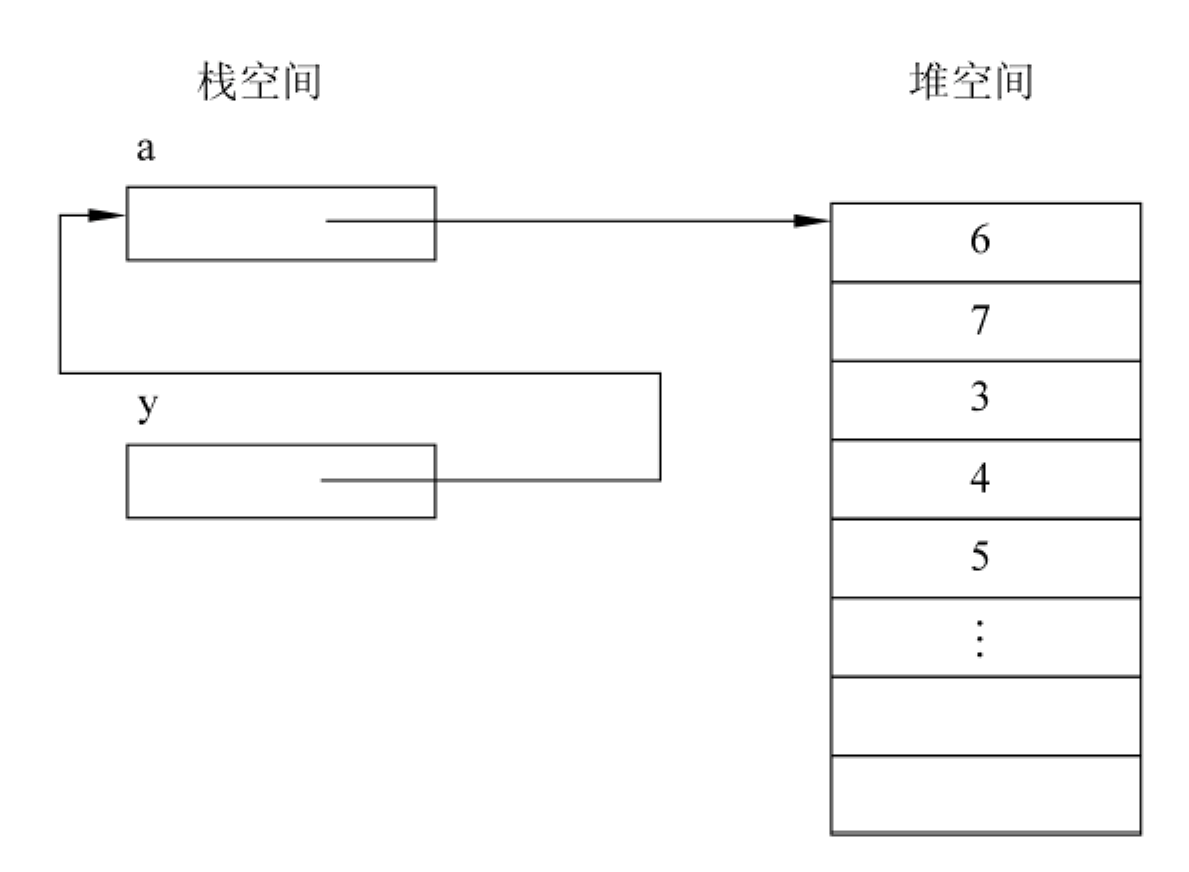


图 6-7　内存示意图(2)

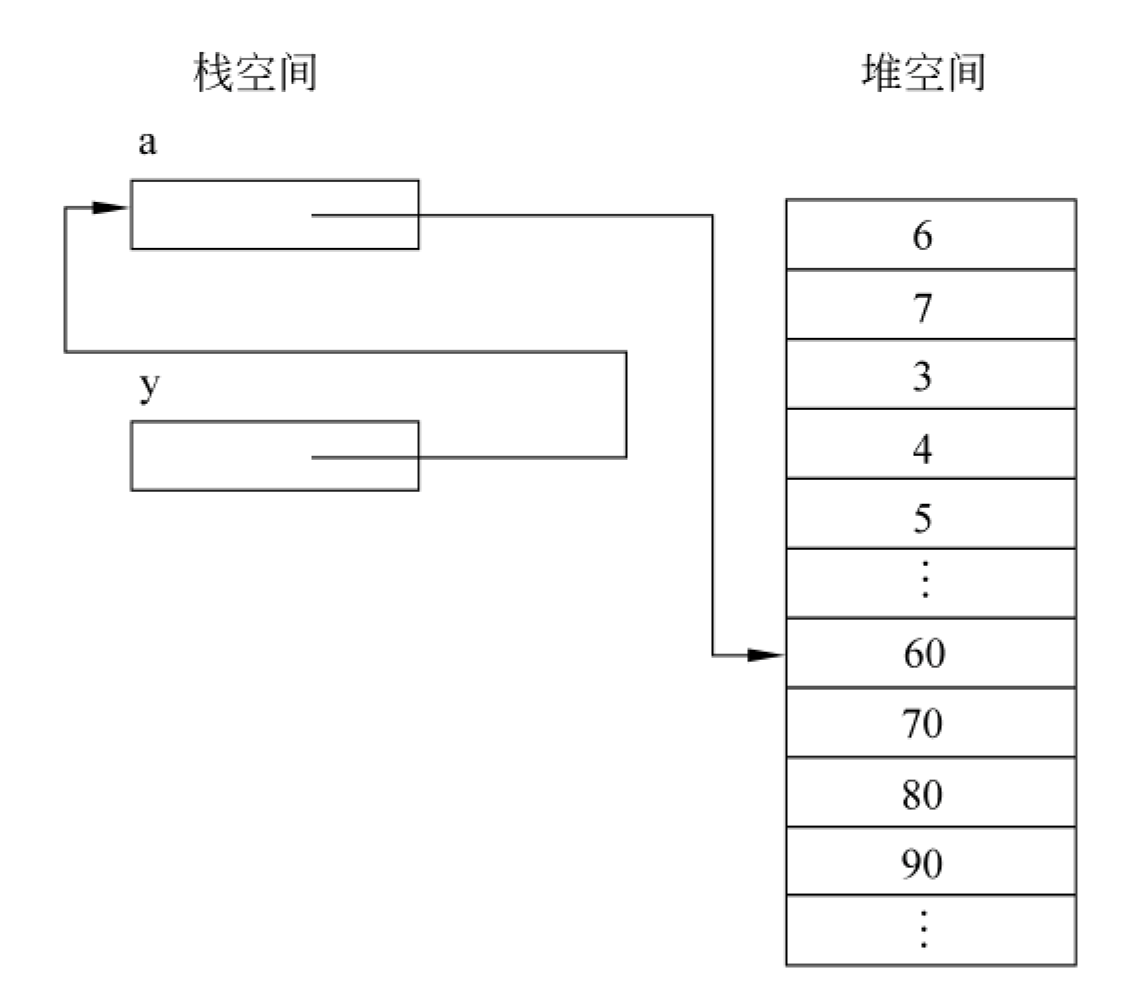


图 6-8　内存示意图(3)

6.4 .NET 框架类库

.NET 框架类库(即 FCL)是生成.NET 应用程序、组件和控件的基础。.NET 框架类库由多个预定义的名字空间构成,而每一个名字空间又是由预定义的多个相关的类组成。名字空间中类型使用点语法命名方案,全名的第一部分(最右边的点之前的内容)是名字空间名。全名的最后一部分是类型名。例如,System.Console 表示 Console 类型,该类型属于 System 名字空间。当通过 using 指令使用 FCL 中的库类时,可不必指定名字空间名。例如,程序中可使用如下指令:

```
using System;
```

程序使用 System 名字空间中的类(如 Console)时,可省略该名字空间名,而直接使用类名。例如,Console.Writeline();与 System.Console.Writeline();等价。

由于 FCL 的名字空间中有许多类,因此在学习 C#时,应多花点时间浏览.NET 文档中的名字空间和类,可以提高编程的效率和质量。

表 6-2 给出了常用的.NET 框架类库中的名字空间及描述。

表 6-2 常用的.NET 框架类库中的名字空间及描述

名字空间	描述
System	所有应用程序使用的一些基本类型
System.Collections	用于管理对象集合,包括常用的集合类型,例如堆栈、队列、散列表等
System.Data System. Data.Linq	用于操纵数据库数据
System.Diagnostics	用于帮助诊断和调试应用程序
System.Drawing	用于操作二维图形,特别是 Windows 窗体应用程序,以及创建 Web 窗体页面中显示的图像
System.EnterpriseServices	用于管理事务、队列组件、对象池、JIT 激活(这里的 JIT 不同于.NET 框架中所言的 JIT 编译,它特指 COM+组件服务,即.NET 内的企业服务中对象的即时激活技术)、安全以及其他一些提高服务器程序中托管代码效能的特性
System.Globalization	用于多国语言支持(National Language Support,NLS),例如字符串比较、格式化以及日历功能
System.IO	I/O 流、遍历目录和文件
System.Ling	支持 LINQ(语言集成查询)
System.Management	通过 Windows 管理设备(Windos Management Instrumentation,WMI)来管理企业中的计算机
System.Net	用于网络通信
System.Reflection	用于查看元数据以及延迟绑定类型及其成员
System.Resurces	用于操作外部数据资源

续表

名 字 空 间	描　　述
System. Runtime. InteropServices	允许托管代码访问非托管操作系统平台中的一些功能，如 COM 组件和 Win32 DLL 内的函数
System. Runtime. Remoting	用于从远程机器上访问类型
System. Runtime. Serialization	用于持久化（Persist）对象实例，以及从一个流（stream）中重新产生对象实例
System. Security	用于操作数据和资源
System. Text	用于以不同的编码方式（如 ASCII 或者 Unicode）来操作文本
System. Threading	用于异步操作以及同步访问资源
System. Xml	用于处理 XML 模式（schema）和数据
System. Web. Services	用于创建 XML Web 服务
System. Web. UI	用于创建 Web 窗体
System. Windows. Forms	用于创建 Windows GUI 应用程序
System. ServiceProcess	用于创建由 SCM 控制的 Windows 服务

6.5　声明的作用域

一个 C# 程序，根据需要可以声明不同类型的实体，如类、方法、属性、变量以及参数等，每个实体都拥有一个实体名称（标识符），使用实体名可以引用这些实体。声明作用域是指使用实体名可直接访问程序代码的范围。下面给出 C# 中几个重要的实体作用域规则：

（1）方法参数声明的作用域是声明所在的方法体。

（2）局部变量声明的作用域从声明点开始，到声明所在块结束为止。

（3）for 语句首部初始化部分出现的局部变量声明的作用域是 for 语句体和首部的其他表达式。

（4）类中成员（如方法、属性、字段）的作用域是整个类体。因此类的非静态方法与属性可以使用类中的任何成员，而静态方法与属性可以使用类中的任何静态成员。

另外，在类的声明中，如果方法中的局部变量或参数与方法所在类中的字段名同名，则在方法中将隐藏该字段，但可通过 this. 字段名的形式使用实例中的字段。

例如，在循环中声明的变量，离开循环后就不起作用了。

```
for(int i=1;i<=10;i++)
{
    int n=i;
    …
}
Console.WriteLine("i={0},n={1}",i,n);
```

以上代码，在 for 循环之后输出循环中声明的变量 i 和 n 将是错误的。因为 for 中声明的变量，是 for 循环的局部变量，该变量只能在 for 中使用。循环结束后，局部变量 i 和 n 就被撤销了。

例 6-12 字段与局部变量的作用域。

程序代码如下：

```
//字段与局部变量的作用域
using System;
namespace CSHARP6_10
{
    public class A
    {
        private int x=1;              //字段变量
        public void f()
        {
            int x=5;                  //局部变量
            Console.WriteLine("输出方法 f的局部变量 x,其值为：{0}",x);
            g();
            h();
            g();
            h();
        }
        public void g()
        {
            int x=8;                  //局部变量
            Console.WriteLine("输出方法 g的局部变量 x,其值为：{0}",x);
            ++x;
        }
        public void h()
        {
            x+=10;
            Console.WriteLine("方法 h使用字段变量 x,其值为：{0}",x);
            ++x;
        }
    }
    public class Test
    {
        public static void Main()
        {
            A t=new A();
            t.f();
        }
    }
}
```

程序开始执行时，Test 类的 Main 方法创建 A 类对象 t(第 34 行)，并调用该对象的 f 方法(第 35 行)。

程序输出如下：

```
输出方法 f的局部变量 x,其值为: 5。
输出方法 g的局部变量 x,其值为: 8。
方法 h使用字段变量 x,其值为: 11。
输出方法 g的局部变量 x,其值为: 8。
方法 h使用字段变量 x,其值为: 22。
```

结果分析：

在 A 类中(05～29 行)，第 10 行声明实例变量 x 并初始化为 1。该实例变量对任何声明局部变量 x 的块(或方法)隐藏。方法 f(08～16 行)声明局部变量 x 并将其初始化为 5。这个局部变量的输出值表明 f 方法中确实隐藏了实例变量 x(其值为 1)。方法 g(17～22 行)声明局部变量 x 并将其初始化为 8，同样隐藏了实例变量 x，两次调用(12～14 行)输出值都是 8。方法 h (23～28 行)中使用实例变量 x，第一次调用(第 13 行)，x 的初值为 1，执行 25 行，x 更新为 11，输出 11，25 行再次更新 x 为 12。第二次调用(第 15 行)，x 的初值为 12，执行 23 行，x 更新为 22，输出 22，25 行再次更新 x 为 23。

6.6　递归

递归是计算机科学的一个重要概念，递归方法是程序设计中有效的方法，采用递归编写程序能使程序变得简洁和清晰。

在数学上，关于递归函数的定义如下：对于某一函数 f(x)，其定义域是集合 A。若对于 A 集合中的某一个值 x0，其函数值 f(x0)由 f(f(x0))决定，那么就称 f(x)为递归函数。

在程序设计语言中，把直接或间接调用自身的方法称为递归方法。这样的递归方法通常必须满足以下两个条件：

(1) 在每一次调用自己时，必须(在某种意义上)更接近于解。

(2) 必须有一个终止处理或计算的准则。

下面通过一个例子来说明递归。用递归程序计算一个非负整数的阶乘，写成 n!。n! 为下列数的乘积：n * (n－1) * (n－2) * … * 1。其中 1!等于 1，0!定义为 1。例如，5!为 5 * 4 * 3 * 2 * 1，即 120。

整数 n 大于或等于 0 时的阶乘可以用下列 for 循环迭代(非递归)计算：

```
fact=1;
for(int i=n;i>=1;i--)
fact=fact * i;
```

通过下列关系可以得到阶乘函数的递归定义：

```
n!=n * (n-1)!
```

例如，5!＝5 * 4!，如下所示：

```
5!=5* 4* 3* 2* 1
5!=5* (4* 3* 2* 1)
5!=5* (4!)
```

求 5!的过程如图 6-9 所示。

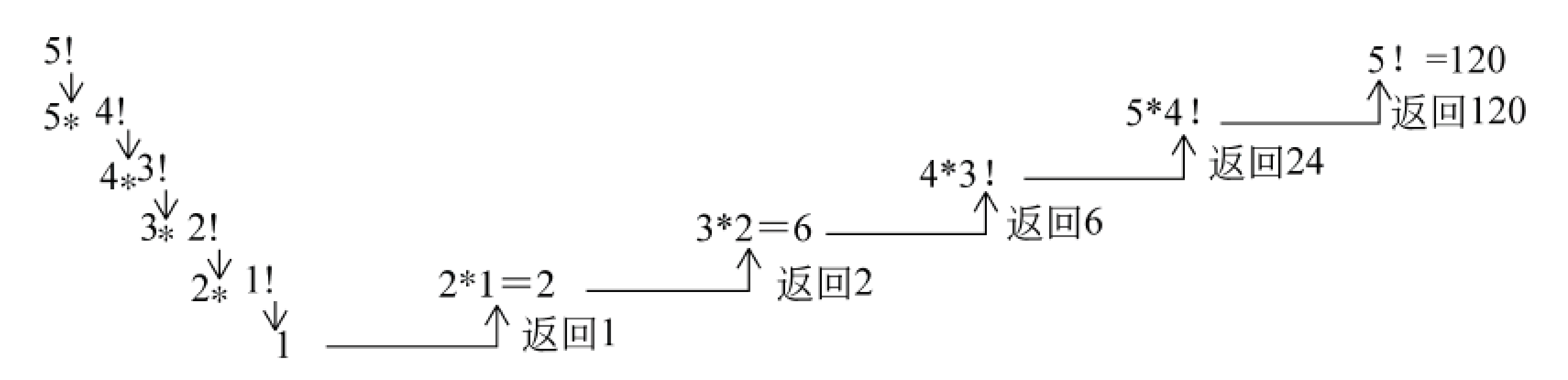


图 6-9 求 5!的过程

图 6-9 显示如何递归调用,直到 1!值为 1,递归终止。

例 6-13 计算并打印 0～10 的整数阶乘。

程序代码如下:

```
//递归方法 Factorial
using System;
namespace CSHARP6_11
{
    public class Factorial
    {
        public static long fact(long n)          //求 n!的递归方法
        {
            if(n<=1)
                return 1;                        //递归结束
            else
                return n* fact(n-1);             //递归调用
        }
        public static void Main()
        {
            for(long n=0;n<=10;n++)
                Console.WriteLine("{0}!={1}",n,fact(n));
        }
    }
}
```

程序运行结果如图 6-10 所示。

特别说明:递归方法对于解决某些复杂问题来说很方便,而且十分强大,但由于频繁使用调用栈(call stack),因此程序执行的效率会大大降低。

例 6-14 枚举所有的布尔变量的组合。当 n=3 时,输出结果如下:

```
true,true,true
true,true,false
```

```
true,false,true
true,false,false
false,true,true
false,true,false
false,false,true
false,false,false
```

如果 n 很大且不用递归，是很难解决这个问题的。

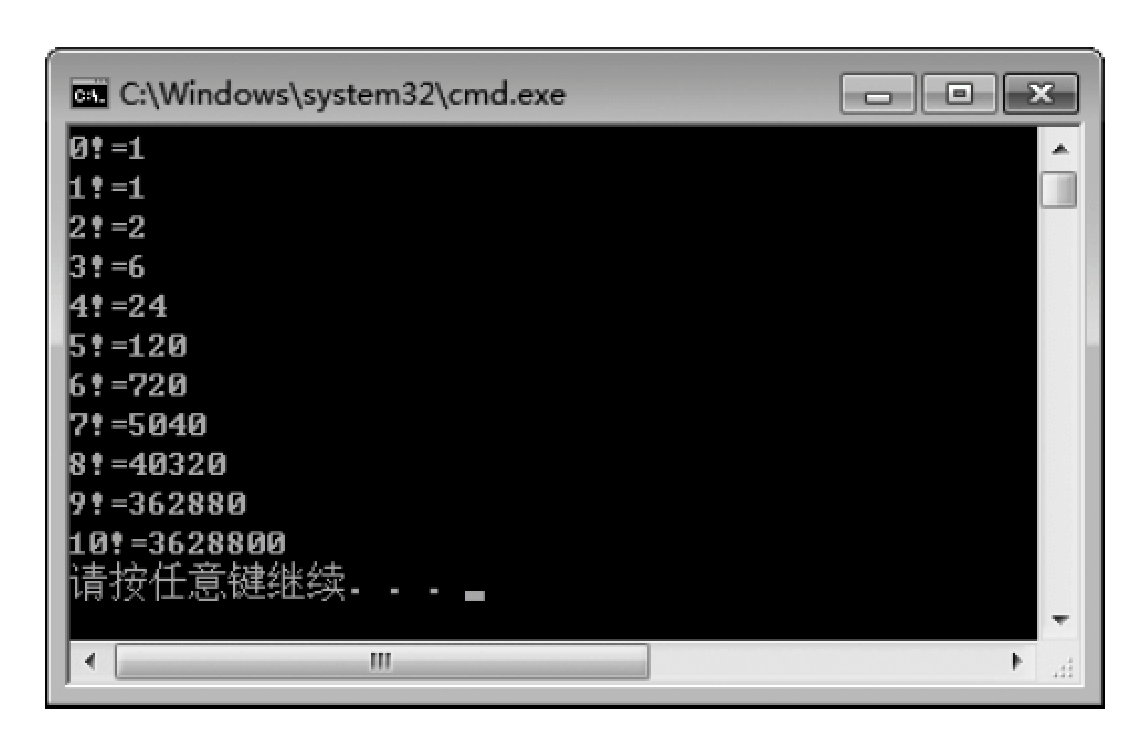


图 6-10　例 6-13 的程序运行结果

程序代码如下：

```
using System;
//使用递归方法枚举所有的布尔变量的组合
namespace CSHARP6_12
{
    //递归方法 Factorial
    class Program
    {
        private static void BooleanCompositions(string partialOutput,
                int counter)
        {
            if (counter< 0)
                Console.WriteLine();
            else if (counter==0)
                Console.WriteLine(partialOutput+"\b ");
            else
            {
                BooleanCompositions(partialOutput+"true,",counter - 1);
                BooleanCompositions(partialOutput+"false,",counter - 1);
            }
        }
        private static void BooleanCompositions(int count)
        {
            BooleanCompositions("true,",count - 1);
```

```
                BooleanCompositions("false,",count - 1);
            }
            static void Main(string[] args)
            {
                int n;
                Console.Write("请输入 n:");
                n= int.Parse(Console.ReadLine());
                BooleanCompositions(n);
            }
    }
}
```

程序运行结果如图 6-11 所示。

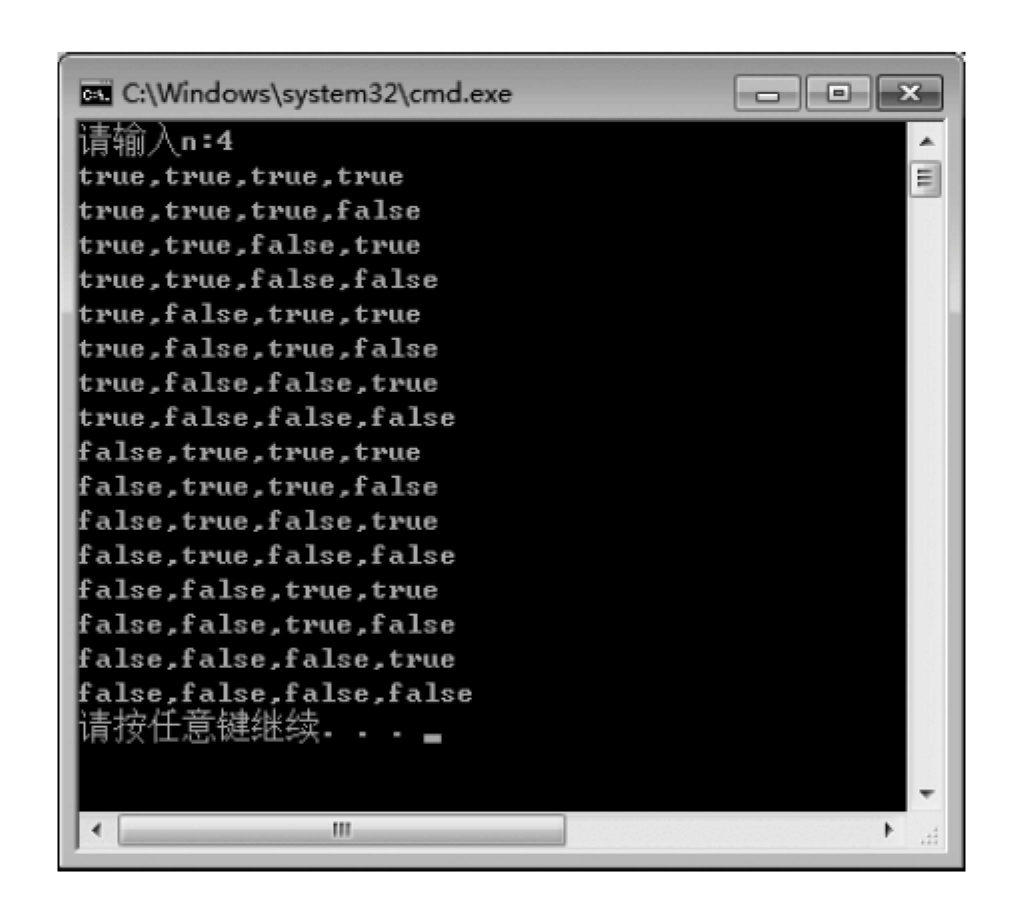


图 6-11 例 6-14 的程序运行结果

习题

1. 阅读程序，判断输出结果。

```
public class samp
{
    public int data= 2;
}
public class Class1
{
    public static void Fun1(samp d)
    {
        d= new samp();
        d.data= 20;
    }
    public static void Fun2(samp d)
```

```
        {
            d.data=20;
        }
        public static void Main(string[] args)
        {
            samp d=new samp();
            Fun1(d);
            Console.WriteLine(d.i);
            Fun2(d);
            Console.WriteLine(d.i);
        }
    }
```

2. 阅读程序,判断输出结果。

```
public class Data
{
    public int i=10;
}
public class Class1
{
    public static void Test1(Data d)
    {
        d.i=100;
    }
    public static void Test2(Data d)
    {
        d=new Data();
        d.i=200;
    }
    public static void Test3(ref Data d)
    {
        d=new Data();
        d.i=300;
    }
    public static void Main(string[] args)
    {
        Data d=new Data();
        Console.WriteLine(d.i);
        Test1(d);
        Console.WriteLine(d.i);
        Test2(d);
        Console.WriteLine(d.i);
        Test3(ref d);
        Console.WriteLine(d.i);
```

```
        }
    }
```

3. 阅读程序，判断“输出参数”使用的正确性。

```
calss App
{
    public static void Useout(int i)
    {
        i=15;
    }
    public static void Main()
    {
        int a=1;
        Useout(a);
        Console.WriteLine("i={0}",a);
    }
}
```

4. 设计一个求解一元二次方程的类，其中包括能够反映一元二次方程特征的域(字段)、方法、属性等，并能合理运用方法的输出参数(out)返回其结果。

5. 编写一个递归函数 power(base，expoent)，它在调用时返回 $base^{expoent}$。

例如，power(3，4)＝3＊3＊3＊3。假设 expoent 是大于或等于 1 的整数。

提示：递归步骤将利用如下关系：

$$base^{expoent}=base.\ base^{expoent-1}$$

并在 exponent 等于 1 时终止递归，因为

$$base^{1}=base$$

6. 编写一个程序，能够显示 1～256 的十进制值对应的二进制、八进制和十六进制值对照表(使用递归算法实现进制转换)。

7. 开发计算机辅助教学程序，教小学生学乘法。程序功能如下：

让用户选择“年级”为 1 或 2。一年级只使用 1 位数乘法；二年级使用 2 位数乘法。要求：用 Random 对象产生两个 1 位或 2 位正整数，然后输入以下问题，例如：“How much is 6 times 7?”然后学生输入答案，程序检查学生的答案。如果正确，则打印“Very good!”，然后提出另一个乘法问题。如果不正确，则打印“No，Please try again.”，然后让学生重复回答这个问题，直到答对。再使用另一个方法产生每个新问题，这个方法在程序开始时和每次用户答对时调用。

第 7 章

面向对象的编程 2

本章继续介绍 OOP 编程中的其他重要概念，包括类成员访问控制、构造函数的重载、类的合成功能、静态类成员、数据抽象与封装等。

7.1　Time 类案例研究

本节通过使用 Time 类的一个完整案例，介绍类的使用中的一些重要概念。

例 7-1　创建两个类，分别是 Time1 类和用于测试 Time1 的 Program 类。在 Program 类的 Main()方法中创建 Time1 类的对象和调用方法。

Time1 类的程序代码如下：

```
 1: public class Time1
 2: {
 3:     private int hour;                              //小时,数据范围 0~23
 4:     private int minute;                            //分,数据范围 0~59
 5:     private int second;                            //秒,数据范围 0~59
 6:     public void SetTime(int h,int m,int s)
 7:     {
 8:         hour= ((h >=0 && h< 24) ? h : 0);          //确保 hour 在有效范围内
 9:         minute= ((m>=0 && m< 60) ? m : 0);
10:         second= ((s>=0 && s< 60) ? s : 0);
11:     }
12:     public string ToUnversalString()               //返回 24 小时制字符串
13:     {
14:         return string.Format("{0:D2}:{1:D2}:{2:D2}",hour,minute,second);
15:     }
16:     public override string ToString()              //返回 12 小时制字符串
17:     {
18:         return string.Format("{0}:{1:D2}:{2:D2} {3}",((hour==0|| hour==
                12)?12:hour %12),minute,second,(hour < 12?"AM":"PM"));
19:     }
20: }
```

代码分析：Time1 类中定义了 3 个 int 类型的专用实例变量(第 3～5 行)，分别表示

时、分、秒，用 24 小时制，类中有 3 个公用方法。Time1 类中没有声明构造函数，因此编译器使用默认的构造函数，每个实例变量隐式使用默认值 0。

第 6～11 行定义了公共的方法 SetTime()，方法中的 3 个参数分别用来提供时、分、秒。第 8～10 行的 3 条赋值语句中使用条件表达式测试每个参数是否在正确的范围内，对于不在有效范围的数据都设置为 0。

第 12～15 行定义了公共的方法 ToUnversalString()。该方法没有参数，用来返回统一的 24 小时制的字符串。字符串长度为 6，时、分、秒分别用两位表示。其中的 return 语句(第 14 行)用 string 类中的一个静态方法 Format 返回一个字符串，包含时、分、秒各两位的值，如果某个值只有 1 位，则在前面添加 0，这是由其中的格式符“D2”确定的。方法 Format 的格式与 Console. Write 方法的格式类似。不同的是方法 Format 返回字符串，而方法 Console. Write 是在控制台窗口显示字符串。

第 16～19 行定义公共方法 ToString()。该方法也没有参数，用来返回统一的 12 小时制的字符串，同样也用 string 类的静态方法 Format 返回一个字符串，将 minute 和 second 格式化为两位，必要时在前面加上 0。第 18 行返回语句中使用表达式“(hour ==0 || hour ==12)?12:hour %12)”确定字符串中 hour 的值;如果该值为 0 或 12 则显示 12，否则显示 1～11 的值。该语句中的另一个表达式“(hour <12?"AM":"PM")”用来确定在返回串的末尾添加“AM”或“PM”。

第 16 行方法的头部使用了修饰关键字 override，表示该方法提供从基类继承的成员的新实现。由 override 声明重写的方法称为重写基方法。重写的基方法必须与 override 方法具有相同的名称。

使用 ToString 方法将对象转换成字符串表示，要在声明 ToString 方法时使用关键字 override(第 16 行)。

TimeTest 类的内容如下：

```
class Program
{
    static void Main(string[] args)
    {
        Time1 time= new Time1();
        Console.WriteLine("初始的时间：");
        Console.Write("\t24 小时制：");
        Console.WriteLine(time.ToUnversalString ());
        Console.Write("\t12 小时制：");
        Console.WriteLine(time.ToString());
        //设置新的时间
        time.SetTime(13,27,6);
        Console.WriteLine("设置的有效时间：");
        Console.Write("\t24 小时制：");
        Console.WriteLine(time.ToUnversalString ());
        Console.Write("\t12 小时制：");
        Console.WriteLine(time.ToString());
```

```
18:             //设置为无效的时间
19:             Console.WriteLine("设置的无效时间：");
20:             time.SetTime(99,99,99);
21:             Console.Write("\t24 小时制：");
22:             Console.WriteLine(time.ToUnversalString ());
23:             Console.Write("\t12 小时制：");
24:             Console.WriteLine(time.ToString());
                //等价于 Console.WriteLine("{0}",time);
25:         }
26: }
```

程序的输出如下。

```
初始的时间：
    24 小时制：00:00:00
    12 小时制：12:00:00 AM
设置的有效时间：
    24 小时制：13:27:06
    12 小时制：1:27:06 PM
设置的无效时间：
    24 小时制：00:00:00
    12 小时制：12:00:00 AM
```

Program 类的代码分析：Program 类用来测试 Time1 类。第 5 行创建 Time1 类的对象并将其赋予局部变量 time，创建对象时没有提供时、分、秒数据。因为 Time1 类中没有声明构造函数，因此 new 调用 Time1 类的默认构造函数，时、分、秒的值都被初始化为 0。

第 6～10 行输出时间，第 8 行使用 24 小时制的格式输出，第 10 行使用 12 小时制的格式输出。

第 12 行调用 Time1 的 SetTime()方法设置新的时间，第 14～17 行再次用两种不同的时间制式输出时间。

第 20 行再次调用 SetTime()方法对时间设置无效的值。在 21～24 行用两种格式输出时间，可以看出，时间显示为 0 点 0 分 0 秒，这是 SetTime()方法对超过有效范围参数的处理结果。

C#中的每个对象都有 ToString 方法，用来返回对象值的字符串表示。调用 Console. Write 带格式项输出对象值时，隐式调用 ToString 方法。

例如，如果第 24 行也可以改写成如下的形式：

```
Console.WriteLine("{0}",time);
```

则输出的结果与原来是一样的，即语句中的 time 隐式调用了 ToString。

7.2　控制对成员的访问

在例 7-1 的 Time1 类中定义了 6 个成员，分别是 3 个变量和 3 个方法。在

Time1Test 中，创建了 Time1 的对象 time，程序中 time 对 Time1 的 3 个方法都进行了访问(调用)。例如第 8、10、12 行)。对象 time 能否访问 Time1 类中的 3 个变量呢？例如使用 time.hour 访问成员 hour。

将 Time1Test 中的 Main()方法改为如图 7-1 所示的内容，即用 time 访问 Time1 类中的 3 个变量。程序运行后，窗口下方显示了 3 个错误，对应这 3 个变量的访问。

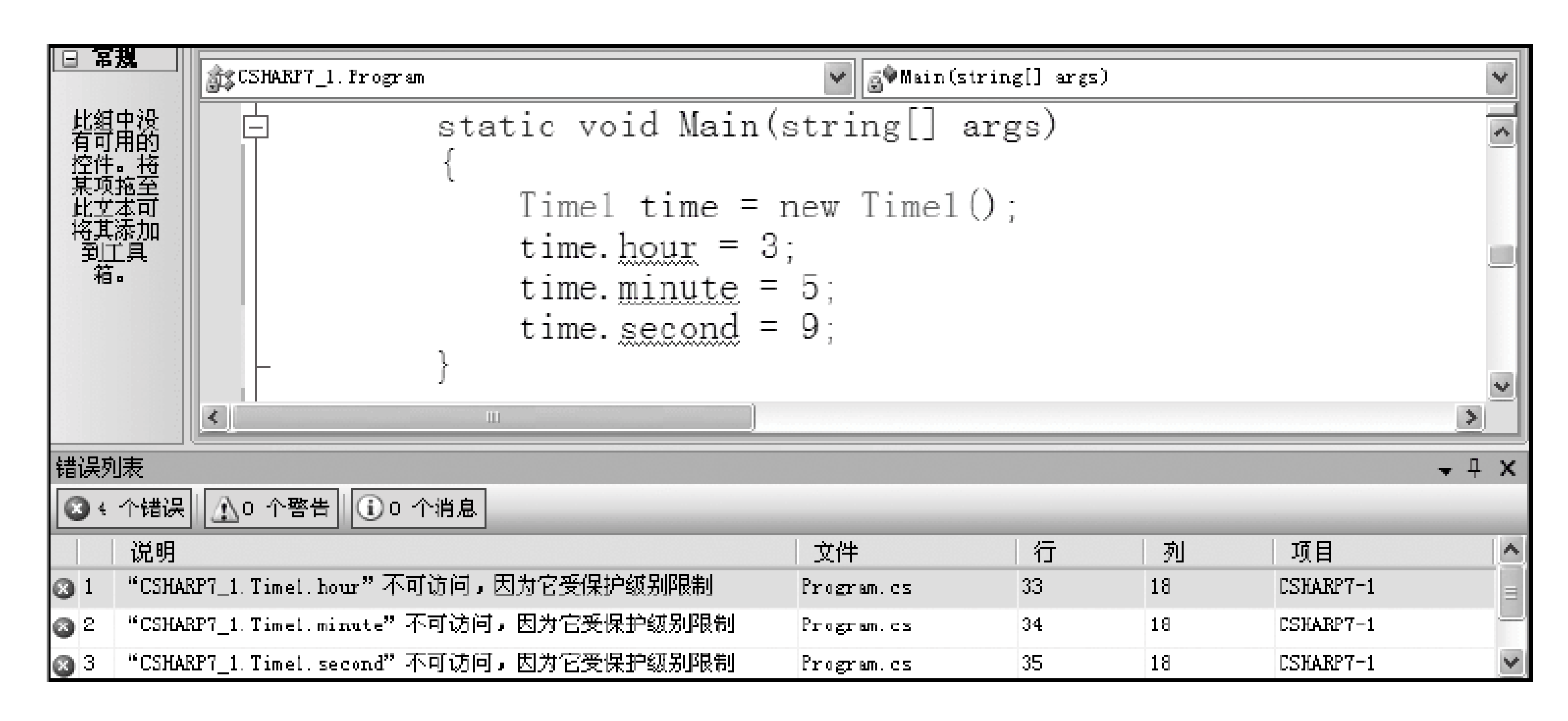


图 7-1　调用 Time1 类中私有成员的出错结果

上面的错误表明，Time1 类中的 3 个变量不允许 Time1 类的对象 time 进行访问。

出错的地方是在 Time1 类的声明中，这 3 个变量的前面都使用了修饰符 private。使用 private 修饰的成员例如专用变量、属性和方法，被称为私有成员。私有成员只能在类内使用，本类和非本类的对象都无法对 private 成员进行访问。

如果类中的成员没有使用访问修饰符，则默认为使用 private。

7.3　用 this 引用访问当前对象的成员

在例 7-1 Time1 类的定义中，第 6～11 行是对类中成员 SetTime 的声明，其声明部分如下：

```
public void SetTime(int hour, int m, int s)
```

该方法中有 3 个参数，即 h、m 和 s，分别用来为 Time1 的 3 个实例变量 hour、minute 和 second 提供数据，例如：

```
hour=h; minute=m,second=s;
```

如果将这 3 个参数的名称改为 hour、minute 和 second，也就是和类中的实例变量同名，这时 3 条赋值语句变成：

```
hour=hour;minute=minute;second=second;
```

在编译程序时会出现下面的提示信息：

对同一变量进行赋值

先确定语句 hour＝hour；中的 hour 是类中的实例变量还是 SetTime 中的形参。在

SetTime 方法的作用域中，隐藏了类中的同名实例变量，只能使用方法中的形参，因此，这 3 条赋值语句中的 3 个变量都指的是形参。

为了在方法中使用与形参同名的实例变量，需要对实例变量使用关键字 this 进行显式的引用。格式是：this. 实例变量名，也可以使用这个格式引用类中的方法名。

事实上，每个对象都可以使用关键字 this 引用自己，这称为 this 引用。在以前的例子中都是隐式地使用 this 调用成员，例如，例 7-1 的类 Time1 中的第 8 条语句：

```
hour= ((h>=0 && h<24) ? h: 0);
```

相当于下面的写法：

```
this.hour= ((h>=0 && h<24) ? h: 0);
```

例 7-2 this 的显式引用。

程序代码如下：

```
using System;

namespace CSHARP7_2
{
    public class SimpleTime
    {
        private int hour;
        private int minute;
        private int second;
        public SimpleTime(int hour,int minute,int second)
        {
            this.hour= hour;
            this.minute= minute;
            this.second= second;
        }
        public string ToUniversalString()
        {
            return string.Format("{0:D2}:{1:D2}:{2:D2}",this.hour,
                   this.minute,this.second);
        }
        public string BuildString()
        {
            return string.Format("{0,24}: {1}\n{2,24}: {3}",
            "this.ToUniversalString()",this.ToUniversalString(),
            "ToUniversalString()",ToUniversalString());
        }
    }
    class ThisTest
    {
```

```
27:         static void Main(string[] args)
28:         {
29:             SimpleTime time= new SimpleTime(13,24,31);
30:             Console.WriteLine(time.BuildString ());
31:         }
32:     }
33: }
```

程序的运行如下：

```
this.ToUniversalString(): 13:24:31
    ToUniversalString(): 13:24:31
```

程序中第 5～24 行定义了类 SimpleTime，类中声明了 3 个实例变量 hour、minute 和 second（第 7～9 行）。第 10～15 行定义了构造函数，构造函数接收 3 个形参为类中的 3 个实例变量赋值，从而初始化对象。此构造函数中使用的 3 个形参名称与类中的 3 个实例变量同名，在构造函数的函数体中有 3 条赋值语句，赋值运算符左侧的变量名前都使用了关键字 this，表明其后的名称是类中的实例变量，而赋值运算符右侧的名称则是方法中的局部变量。这三条语句都是显式地引用实例变量，这里的显式写法是必需的。

第 16～19 行定义了 ToUniversalString()方法，用来输入显示时间的字符串。该方法与例 7-1 中的方法作用是一样的，但方法中对实例变量也使用了显式的引用（第 18 行），但是这个方法中的显式引用不是必需的，因为在这个方法中，没有重名的变量，出现的 3 个变量都是类中的实例变量。在例 7-1 中该方法就没有使用显式引用。

第 20～23 行定义了 BuildString()方法，该方法同样是返回字符串。在 return 语句中两次调用了 ToUniversalString()方法，不同的是，第一次调用是显式 this 引用，第二次是隐式 this 引用，但调用的效果都是一样的，输出结果中的两行是相同的。

第 25～32 行定义 ThisTest 类，该类用来测试 SimpleTime 类。第 29 行创建 SimpleTime 类的对象并调用其构造函数。第 30 行调用 BuildString()方法显示结果。

只有在方法中包含与字段同名的局部变量时，才显式地使用 this 来引用类中被隐藏的实例变量，通常在方法中的参数或局部变量应避免和类中的实例变量同名。

7.4 构造函数与析构函数

7.4.1 重载构造函数

在类中定义的构造函数，可以对类对象进行初始化。类的构造函数也可以重载，只要这些构造函数的参数个数、类型和顺序不同即可以重载，这样可以对类的对象用不同的方式进行初始化。

例 7-3 构造函数的重载。

在 Time 类中定义 4 个重载的构造函数，每个函数中的参数个数不同，在 Test 类中进行测试，程序代码如下：

```
using System;

namespace CSHARP7_3
{
    public class Time
    {
        private int hour;
        private int minute;
        private int second;
        public Time()
        {
            hour=minute=second=0;
        }
        public Time(int h)
        {
            hour=h;
            minute=second=0;
        }
        public Time(int h,int m)
        {
            hour=h;
            minute=m;
            second=0;
        }
        public Time(int h,int m,int s)
        {
            hour=h;
            minute=m;
            second=s;
        }
        public override string ToString()
        {
            return string.Format("{0:D2}:{1:D2}:{2:D2}",hour,minute,
                second);
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            Time time1=new Time();
            Console.WriteLine(time1);
            Time time2=new Time(1);
            Console.WriteLine(time2);
```

```
44:                 Time time3= new Time(1,32);
45:                 Console.WriteLine(time3);
46:                 Time time4= new Time(1,32,45);
47:                 Console.WriteLine(time4);
48:             }
49:     }
50: }
```

程序的运行结果如下：

```
00:00:00
01:00:00
01:32:00
01:32:45
```

程序分析：第 5～35 行定义了 Time 类，该类中有 3 个实例变量（第 7～9 行），分别表示时、分、秒。

类中有 4 个构造函数，参数个数分别是 0、1、2、3 个。第 10～13 行定义第 1 个构造函数，函数中没有参数，是无参构造函数，函数体中对类的 3 个实例变量都赋值为 0。

第 14～18 行定义第 2 个构造函数，该函数中有一个实参，函数体中将这个形参赋给变量 hour，另外两个变量赋值为 0。

第 19～24 行定义第 3 个构造函数，函数中有 2 个形参，函数体中将这两个形参分别赋给变量 hour 和 minute，而 second 变量则赋值为 0。

第 25～30 行定义第 4 个构造函数，函数中有 3 个形参，在函数体中将这 3 个形参按顺序分别赋给 hour、minute 和 second 这 3 个变量。

第 31～34 行重写 ToString()方法输入对象的字符串。

在 Test 类中分别定义了 4 个 Time 类的变量 time1、time2、time3 和 time4，并用创建的对象进行赋值，然后隐含调用 Time 的 ToString 方法显示时间信息。但是，每次创建对象时给出的参数个数不同，分别是 0～3 个。例如创建第 1 个对象时没有参数，所以自动调用第 10 行的构造函数，hour、minute 和 second 这 3 个变量的值都是 0，第 1 行输出 0 点 0 分 0 秒。

创建第 2 个对象时只有一个参数，所以自动调用第 14 行的第 2 个构造函数，并将这个参数 1 赋给了变量 hour，其他两个变量赋值为 0，第 2 行输出 1 点 0 分 0 秒。

同样地，创建后两个对象时分别调用了另外两个构造函数。

本例的前 3 个构造函数中也可以调用第 4 个构造函数。例如，可以将第 10～24 行 3 个构造函数写成如下的形式：

```
public Time():this(0,0,0) { }
public Time(int h):this(h,0,0) { }
public Time(int h,int m):this(h,m,0) { }
```

这里的调用使用了前面介绍的 this 关键字。

7.4.2 默认构造函数

每个类中至少要有一个构造函数,如果在类的声明中没有提供构造函数,则编译器生成默认的构造函数,该构造函数调用时不使用任何参数。

例 7-4 调用默认的构造函数。

程序中定义的 Time 类中没有定义构造函数,在 Test 类中测试这个类,程序代码如下:

```
using System;

namespace CSHARP7_4
{
    public class Time
    {
        private int hour;
        private int minute;
        private int second;
        public override string ToString()
        {
            return string.Format("{0:D2}:{1:D2}:{2:D2}",hour,minute,second);
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            Time time1= new Time();
            Console.WriteLine(time1);
        }
    }
}
```

程序的运行结果如下:

```
00:00:00
```

在 Test 类的 Main()方法中,创建了一个 Test 类的对象 Time1,但没有使用参数对其进行初始化。由于 Test 类中没有一个构造函数,因此程序自动调用默认的构造函数,但变量没有被赋值,一直保持默认值 0,所以输出结果为 0 时 0 分 0 秒。

7.4.3 内存回收与析构函数

创建每个对象都要使用各种系统的资源,例如内存资源。这些资源保留给对象使用,直到其显式地释放。如果管理资源的对象失去了所有的引用,但还没有释放资源,则程序就不能再访问和释放这个资源,这样,就产生了资源泄漏。

为了避免资源泄漏，公共语言运行环境（CLR）进行自动的内存管理，用垃圾回收器释放某个对象不再需要的内存，以便其他对象可以使用这个内存。

为实现管理，每个对象都有一个特殊的成员，称为析构函数。该函数由垃圾回收器调用，在垃圾回收器释放对象的内存之前用于终止对象前的管理工作。当垃圾回收器调用了析构函数之后，对象就可以回收内存了。

通常这个回收过程是自动进行的，但是，也有可能发生其他类型的资源泄漏。例如，程序打开了一个磁盘文件并修改了文件的内容，如果没有关闭文件，则其他程序无法修改这个文件。

在默认情况下，编译器自动生成空的析构函数，因此在 C# 中不允许定义空的析构函数。析构函数的一般形式如下：

```
~类名 ()
{
    语句
}
```

关于析构函数要注意如下的问题：

(1) 析构函数命名时，在类名之前加上“～”符号。

(2) 只能在类中定义析构函数，并且一个类中只能有一个析构函数。

(3) 析构函数不能继承或重载。

(4) 析构函数声明时没有修饰符和参数。

(5) 析构函数不能显式地调用，由系统在释放对象时自动调用。

例 7-5 自动调用析构函数。

程序中定义了 Time 类，类中有析构函数，在 Test 类的 Main()方法中进行测试。程序代码如下：

```
using System;

namespace CSHARP7_5
{
    public class Time
    {
        private int hour;
        private int minute;
        private int second;
        public Time()
        {
            hour= minute= second= 0;
        }
        ~Time()
        {
            Console.WriteLine("Time的析构函数被自动调用");
```

```
17:            }
18:            public override string ToString()
19:            {
20:                return string.Format("{0:D2}:{1:D2}:{2:D2}",hour,minute,second);
21:            }
22:        }
23:        class Test
24:        {
25:            static void Main(string[] args)
26:            {
27:                Time time1= new Time();
28:                Console.WriteLine(time1);
29:            }
30:        }
31: }
```

程序的运行结果如下：

```
00:00:00
Time的析构函数被自动调用
```

程序中第 14～17 行定义了析构函数，函数中只有一条输出语句，在 Test 类的 Main()方法中，创建了一个 Time 类的对象 time1 并自动调用了无参构造函数(第 27 行)。第 28 行输出了该对象的字符串。遇见第 29 行的“}”时，Main()方法结束，同时对象 time1 也被释放，释放时自动调用析构函数，这样，输出结果中第 2 行显示了析构函数中的输出信息。

对比类的构造函数：构造函数在创建对象时自动调用，析构函数在撤销一个对象时自动调用；构造函数允许重载，析构函数不允许重载。

7.4.4 对象初始化器

在 Visual C# 2008 以后的各个版本中，有一个新的工具，这就是对象初始化器，用来在创建对象时在同一语句中将其属性初始化，这适用于类中没有构造函数的情形。

例 7-6 使用对象初始化器初始化新建的对象。

在类 Time 中添加 3 个属性访问器，每个访问器中都包含 get 访问器和 set 访问器。在 Test 类的 Main()方法中使用对象初始化器进行测试。程序代码如下：

```
using System;

namespace CSHARP7_6
{
    public class Time
    {
        private int hour;                    //这里变量名的第 1个字母为小写
        private int minute;
```

```
        private int second;
        public int Hour                    //这里属性名的第 1 个字母为大写
        {
            get{ return hour; }
            set{ hour= (value>=0 && value<24)?value:0; }
        }
        public int Minute
        {
            get { return minute; }
            set { minute= (value>=0 && value<60)?value:0;}
        }
        public int Second
        {
            get { return second; }
            set { second= (value>=0 && value<60)?value:0;}
        }
        public void SetTime(int h,int m,int s)
        {
            hour= ((h>=0 && h<24) ? h : 0);    //确保 hour 在有效范围内
            minute= ((m>=0 && m<60) ? m : 0);
            second= ((s>=0 && s<60) ? s : 0);
        }
        public override string ToString()
        {
            return string.Format("{0:D2}:{1:D2}:{2:D2}",hour,minute,second);
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            Time t1=new Time{ Hour=3,Minute=5,Second=6 };        //对象初始化器
            Console.WriteLine (t1);
            Time t2=new Time{ Hour=3,Second=6 };        //默认调用 ToString()方法
            Console.WriteLine (t2);
            Time t3=new Time{ Second=6 };
            Console.WriteLine(t3);
        }
    }
}
```

程序的运行结果如下：

```
03:05:06
03:00:06
```

```
00:00:06
```

程序分析：第 5～35 行定义类 Time。第 10～14 定义了类中的一个属性 Hour，用来访问 hour 变量，其中的 get 访问器返回变量 hour，set 用于设置变量 hour 的值，其中的条件表达式保证该变量在正确的范围内(0～23)。

第 15～19 行、20～24 行定义了另外两个属性 Minute 和 Second，分别用来访问变量 minute 和 second。

第 25～30、31～34 行定义的方法与前面的例题相同。

第 36～47 行定义了类 Test。该类中定义了 3 个 Time 类的对象，输出结果中的每一行显示了这 3 个对象的字符串，第 40 行语句 new Time 后面的内容：

```
{ Hour= 3,Minute= 5,Second= 6 };
```

这个格式就是对象初始化器，用一对花括号括起来的是对各个属性的赋值。由于每个属性的 set 访问器可以给变量赋值，因此，对象初始化器中给各个属性的赋值都分别传递给了变量。例如属性 Hour＝3 最终传递给变量 hour，另外两个也是一样。

这样，第 1 行输出 3 点 5 分 6 秒。

使用对象初始化器时，要注意下面的问题：

(1) 初始化器中对各个属性的赋值没有顺序的要求，第 40 行中 new Time 后面的内容也可以写成下面的顺序：

```
{ Hour= 3,Second= 6,Minute= 5 };
```

(2) 初始化器中没有要求对所有的属性赋值，例如第 42、44 行定义对象 t2 和 t3 时只分别给 2 个属性和 1 个属性赋值。

7.5　合成

一个类中的成员也可以是其他类的对象，这个功能称为合成，也称为“有”的关系，合成也是软件复用的一种方式。

例 7-7　职工档案管理系统。

分别定义工资类 EmpSalary、日期类 Date、职工类 Employee。工资类包括基本工资、岗位津贴、房租、电费和水费。日期类包括年、月、日三种数据成员。职工类包括工作部门、姓名、出生日期、职务、参加工作时间和工资。其中的职务类型为枚举类型，有经理、工程师、职员和工人四种类型。最后在类 My 的 Main()方法中进行验证。

程序代码如下：

```
using System;

namespace CSHARP7_7
{
    class EmpSalary
```

```
{                                          //定义工资类
    public float Wage;                     //基本工资
    public float Subsidy;                  //岗位津贴
    public float Rent;                     //房租
    public float CostOfElec;               //电费
    public float CostOfWater;              //水费
    public float RealSum
    {                                      //计算实发工资
        get
        {
            return Wage+ Subsidy - Rent - CostOfElec - CostOfWater;
        }
    }
}
public enum Position
{                                          //枚举类型,定义不同的职务
    MANAGER,                               //经理
    ENGINEER,                              //工程师
    EMPLOYEE,                              //职员
    WORKER                                 //工人
}
class Date                                 //定义日期类
{
    int day,month,year;
    public Date(int yy,int mm,int dd)
    {
        init(yy,mm,dd);
    }
    public Date(Date x)
    {
        init(x.year,x.month,x.day);
    }
    public void init(int yy,int mm,int dd)
    {
        month= (mm >=1 && mm <=12) ? mm : 1;
        year= (yy >=1900 && yy <=2100) ? yy : 1900;
        day= (dd >=1 && dd <=31) ? dd : 1;
    }
    public void Printymd()                 //按年月日顺序输出日期
    {
        Console.WriteLine("{0}- {1}- {2}",year,month,day);
    }
}
```

```
    class Employee
    {                                       //定义职工类
        string Department;                  //工作部门
        string Name;                        //姓名
        Date Birthdate;                     //出生日期
        Position EmpPosition;               //职务,是类 Position 的对象
        Date DateOfWork;                    //参加工作时间,是类 Date 的对象
        EmpSalary Salary;                   //工资
        public Employee(string Depart,string Name,Date tBirthdate,
                Position nPosition,Date tDateOfWork)
        {
            this.Department= Depart;
            this.Name= Name;
            this.Birthdate= new Date(tBirthdate);   //创建 Date 类的对象
            this.EmpPosition= nPosition;
            this.DateOfWork= new Date(tDateOfWork); //创建 Date 类的对象
            this.Salary= new EmpSalary();                   //创建 EmpSalary 类的对象
        }
        public void SetSalary(float wage,float subsidy,float rent,float
               elec,float water)
        {
            Salary.Wage= wage;
            Salary.Subsidy= subsidy;
            Salary.Rent= rent;
            Salary.CostOfElec= elec;
            Salary.CostOfWater= water;
        }
        public float GetSalary
        {
            get
            {
                return Salary.RealSum;
            }
        }
        public void ShowMessage()                   //打印职工信息
        {
            Console.WriteLine("部门:{0}",Department);
            Console.WriteLine("姓名:{0}",Name);
            Console.Write("出生日期:");
            Birthdate.Printymd();
            Console.Write("职务:");
            switch (EmpPosition)
            {
```

```
                    case Position.MANAGER:
                        Console.WriteLine("经理"); break;
                    case Position.ENGINEER:
                        Console.WriteLine("工程师"); break;
                    case Position.EMPLOYEE:
                        Console.WriteLine("职员"); break;
                    case Position.WORKER:
                        Console.WriteLine("工人"); break;
                }
                Console.Write("工作日期: ");
                DateOfWork.Printymd();
                Console.WriteLine("工资: {0}",GetSalary);
                Console.WriteLine("-----------------------------------");
            }
        }
        class My
        {
            static int Main()
            {
                //第一个职工数据
                Date birthdate=new Date(1980,5,3);
                Date workdate=new Date(1999,7,20);
                Employee Empl=new Employee("销售处","张弓长",birthdate,
                        Position.ENGINEER,workdate);
                Empl.SetSalary(3000,2000,1000,50,20);
                Empl.ShowMessage();
                //第二个职工数据
                birthdate.init(1979,4,8);
                workdate.init(2002,3,1);
                Employee Emp2=new Employee("项目部","李木子",birthdate,
                        Position.MANAGER,workdate);
                Emp2.SetSalary(2500,2900,1500,50,20);
                Emp2.ShowMessage();
            }
        }
}
```

程序的运行结果如下：

部门：销售处
姓名：张弓长
出生日期：1980-5-3
职务：工程师
工作日期：1999-7-20

```
工资：3930
---------------------------------
部门：项目部
姓名：李木子
出生日期：1979-4-8
职务：经理
工作日期：2002-3-1
工资：3830
---------------------------------
```

代码分析：第 8～19 行定义工资类 EmpSalary，第 20～26 行定义枚举类型 Position，第 27～48 行定义日期类 Date。

第 49～104 行定义职工类 Employee，第 51～56 行定义了该类中的 6 个变量，其中变量职务 EmpPosition 是类 Position 的对象（第 54 行），变量 DateOfWork 则是类 Date 的对象（第 55 行），这两行就是类的合成功能。

第 57～65 是类 Employee 中的构造函数，由于 Employee 类的成员中含有其他类的对象，因此，构造函数的函数体中也创建了其他类的对象（第 61、63、64 行）。

在 My 类的 Main()方法中，创建了两个 Date 类的对象 birthdate 和 workdate、两个 Employee 的对象 Emp1 和 Emp2 即两个员工，最后显示输出这两个员工的信息。

7.6　readonly 实例变量

在类中可以对某个实例变量使用 const 进行修饰，并且在声明时初始化为常量值。这时该变量变成了常量，在该类的各个对象中都使用相同的值而且不能修改。

如果要在类中定义这样的变量，其值在构造函数中进行初始化，初始化后就不能修改，但不同的对象调用构造函数时可以将该变量初始化为不同的值，这样的变量可以使用 readonly 关键字进行修饰。例如，下面的语句定义了一个只读的变量 INCREMENT：

```
private readonly int INCREMENT;
```

只读变量的命名习惯上全部为大写字母，在声明语句中可以对实例变量进行初始化，但通常是由类的每个构造函数初始化，构造函数可以多次对只读实例变量赋值。

声明为 const 的成员在编译时初始化，而声明为 readonly 的成员在运行时初始化。

例 7-8　只读实例变量的使用。

程序中定义了两个类，分别是 Increment 和 IncrementTest 类。Increment 中含有 int 类型的 readonly 实例变量，在 IncrementTest 类中测试只读变量的使用。程序代码如下：

```
using System;

namespace CSHARP7_9
{
    public class Increment
```

```
    {
        private readonly int INCREMENT;        //只读变量,保存递增的增量
        private int total=0;
        public Increment(int incrementValue)
        {
            INCREMENT=incrementValue;
        }
        public void AddIncrementToTotal()
        {
            total+=INCREMENT;                  //累加到 total 中
        }
        public override string ToString()
        {
            return string.Format("total={0}",total);
        }
    }
    class IncrementTest
    {
        static void Main(string[] args)
        {
            Console.WriteLine("======创建第 1个对象======");
            Increment incrementer1=new Increment(5);
            Console.WriteLine("递增之前:{0}",incrementer1);
            for (int i=1; i<=3; i++)
            {
                incrementer1.AddIncrementToTotal();
                Console.WriteLine("第{0}次递增之后:{1}",i,incrementer1);
            }
            Console.WriteLine("\n======创建第 2个对象======");
            Increment incrementer2=new Increment(6);
            Console.WriteLine("递增之前:{0}",incrementer2);
            for (int i=1; i<=3; i++)
            {
                incrementer2.AddIncrementToTotal();
                Console.WriteLine("第{0}次递增之后:{1}",i,incrementer2);
            }
        }
    }
}
```

程序的运行结果如下:

```
======创建第 1个对象======
递增之前:total=0
```

```
第 1次递增之后：total=5
第 2次递增之后：total=10
第 3次递增之后：total=15

======创建第 2个对象======
递增之前：total=0
第 1次递增之后：total=6
第 2次递增之后：total=12
第 3次递增之后：total=18
```

代码分析：第 5～21 行定义了 IncrementTest 类，该类中有一个 readonly 类型的变量 INCREMENT(第 7 行)，该变量没有在声明时初始化。还有一个变量是 total，用来保存累加的值。第 9～12 行定义构造函数，在该函数中对 INCREMENT 进行了初始化，其值由形参传递过来，也就是说，每次创建对象时都可以对 INCREMENT 用不同的值进行初始化。

如果对 readonly 类型的变量在声明时进行初始化，其效果和使用 const 修饰是一样的。

如果在构造函数中添加一条语句：

```
INCREMENT++;
```

则在编译时会产生下面的错误信息：

```
无法对只读的字段赋值(构造函数或变量初始值指定项中除外)
```

第 13～16 行定义方法 AddIncrementToTotal()用来进行累加，并将累加结果保存在变量 total 中。

第 22～43 行定义了 IncrementTest 类，该类的 Main()方法中测试 Increment 类。第 27 行创建了 Increment 类的第 1 个对象 incrementer1，初始化时的参数为 5，这样，构造函数中将 INCREMENT 变量初始化为 5。第 28 行显示在进行递增运算之前，total 变量的值为 0。第 29～33 行通过循环语句 3 次调用 AddIncrementToTotal()方法对变量 total 进行了 3 次递增操作，每次的增量为 5，所以每次递增后的结果分别是 5、10、15。

第 35 行创建了第 2 个对象 incrementer2，初始化时的参数为 6。这样，构造函数中将 INCREMENT 变量初始化为 6，第 35 行显示在进行递增运算之前，total 变量的值为 0。第 37～41 行调用 AddIncrementToTotal()方法对变量 total 进行了 3 次递增操作，每次的增量为 6，所以每次递增后的结果分别是 6、12、18。

7.7　数据抽象与封装

类是对数据和处理数据的方法(函数)的封装，封装把类的属性和方法看成一个密不可分的整体，从而使类具有明确的独立性，这样能够完整地描述并对应于一类具体的事物。封装是面向对象程序设计的基本特征之一。

类的封装对于类的使用者隐藏了实现的细节，这种描述类的功能而不管其实现细节的方法称为数据抽象。

C＃中面向对象编程的主要活动就是生成自己的新的数据类型(即类)和表达这些数据类型之间的相互作用。显然，C＃等面向对象的语言提升了数据的重要性，这样就要求更规范地组织数据，这里的规范化就是抽象数据类型(ADT)，ADT 可以进一步改善程序的开发过程。

下面看一个简单的数据抽象。C＃中的数据类型 int 与数学中的整数有联系，但并不完全是数学中的整数，它是整数的抽象表示。在计算机中的 int 其长度是有限的。例如 C＃中的 int 是－2147483648～2147483647 之间的整数。如果计算结果超过这个范围，就会产生错误。

ADT 的概念包含两个方面，数据表达和该数据允许的操作。例如 int 类型的数据在 C＃中定义了加、减、乘、除和求余操作。

关于 ADT 的其他例子可以参考本书数据结构一章中有关栈和队列的内容。

7.8 Class View 与 Object Browser

本节介绍 Visual Studio 环境中的两个重要窗格。这两个窗格对面向对象程序设计很有帮助，分别是 Class View(类视图)和 Object Browser(对象浏览器)。

1. Class View

从"视图"(View)菜单中选择"类视图"(Class View)，Visual Studio 的窗口中显示类视图窗口，如图 7-2 所示。

图中显示的是例 7-2 中的类。视图中使用层次结构显示，窗格上方显示的是该工程中的类，图中显示 CSHARP7-1 项目中的有两个类，分别是 Program 和 Time1。

单击某个类，则窗格下方显示出该类的各个成员，图中显示 Time1 类的 6 个成员，分别是 3 个方法 SetTime、ToString、ToUniversalString，3 个实例变量 hour、minute、second。不同类型的成员其名称前面用不同的图标显示，例如实例变量前面显示带有锁的图标表示实例变量为专用的。

图 7-2 类视图

2. Object Browser

从"视图"(View)菜单中选择"其他窗口"(Other Windows)，然后选择"对象浏览器"(Object Browser)，可以打开对象浏览器，如图 7-3 所示。其中显示 C＃库中的所有类，可以通过它了解某个类提供的功能。

该窗格分为 3 个部分，左边显示所有的类，包括系统定义的类和用户自定义的类。单击某个类，窗格右边的上方显示了该类的所有成员，图 7-2 中显示的是用户自定义的类

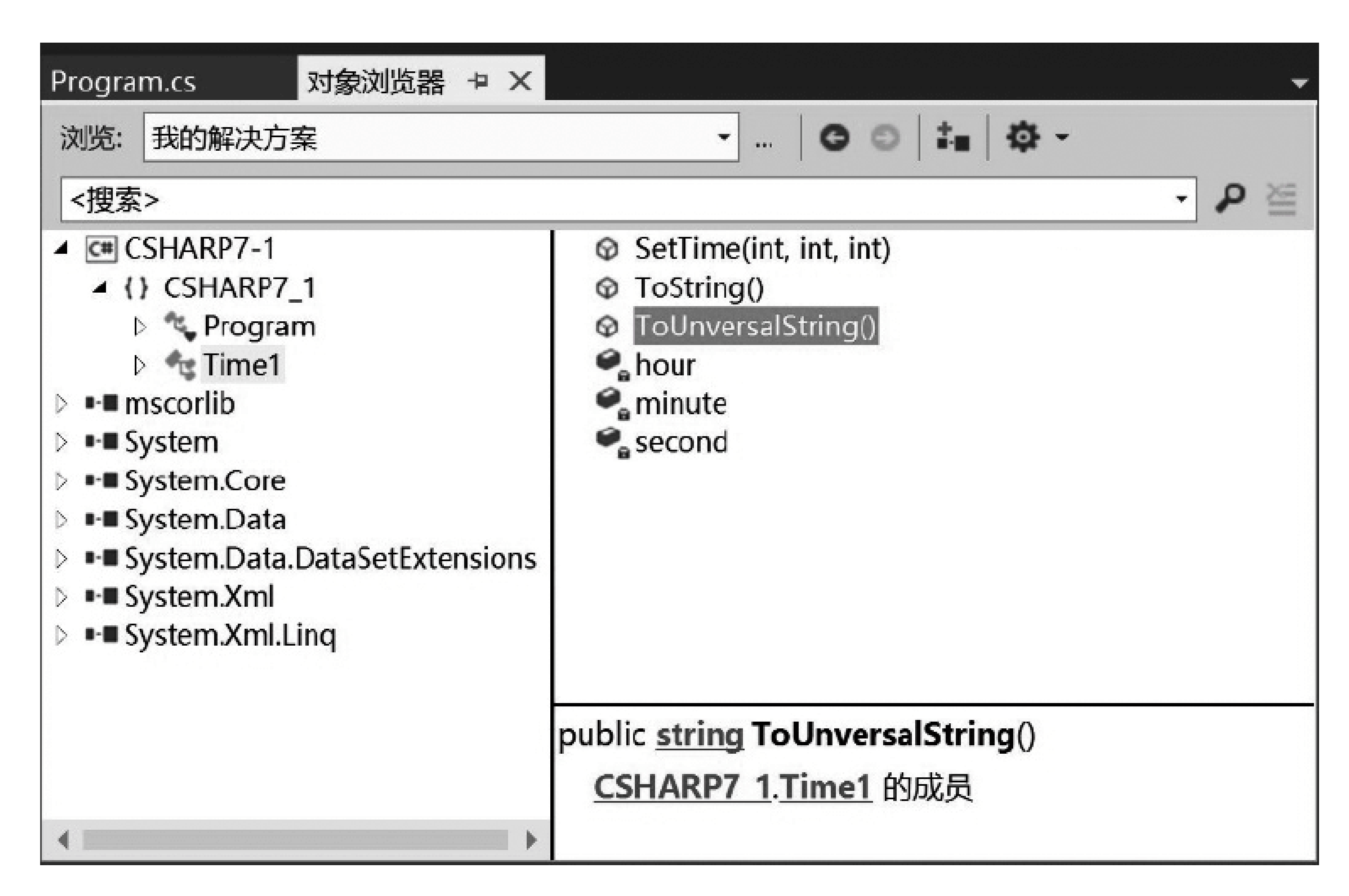


图 7-3　对象浏览器

Time1。单击类中的某个成员，则右边的下方显示该成员的具体描述。

使用该窗格可以方便学习 C# 中的类及其方法。

习题

1. 定义 Rectangle 类(长方形)，类中的两个属性 length 和 width 默认值为 1，其只读属性计算长方形的周长(Perimeter)和面积(Area)。为该类的 length 和 width 设置 set 和 get 方法，在 set 方法中验证 length 和 width 是在 0.0～20.0 之间(不包括这两个数)的浮点数，再定义另一个类测试 Rectangle 类。

2. 修改例 7-3，将时间表示为从午夜 0 点算起的秒数，不使用 3 个整数 hour、minute 和 second，类的功能不变，然后使用相同的类 Test 进行测试。

3. 定义 SavingsAccount 类，类中有如下成员：

- 静态变量 annualInterestRate 存储所有账户持有者的年利率。
- 类的每个对象包含一个专用实例变量 savingBalance，表示该存款账户当前的金额。
- 定义 CalculateMonthlyInterest 方法，将 annualInterestRate 与 savingBalance 相乘后除以 12 得到月利息，将这个月利息加进 savingBalance 中。
- 定义静态方法 ModifyInterestRate，将 savingBalance 设置为新值。

定义另一个类测试 SavingsAccount 类。创建两个 SavingsAccount 的对象 saver1 和 saver2，结余分别为 2000.0 美元和 3000.0 美元。将 annualInterestRate 设置为 4%，计算月利息并打印两个账户的新结余，然后将 annualInterestRate 设置为 5%。

4. 修改例 7-1 中的 Time 类，添加以下 3 个方法：

- Tick 方法将对象中存放的时间递增一秒。
- IncrementMinute 方法递增分钟。

- IncrementHour 方法递增小时。

在另一个类中测试 Time 类，分别调用这 3 个方法，要保证测试下列情况：

- 递增到下一分钟。
- 递增到下一小时。
- 递增到下一天（从 11:59:59 PM 到 12:00:00 AM）。

5. 创建 Date 类，要求具有下面的功能：

（1）用以下各种格式输出日期：

```
MM/DD/YYYY
June 14,1992
DDD YYY
```

（2）用重载构造函数创建 Date 对象，用上面的各种格式初始化日期。第一种情况下构造函数接收 3 个整数；第 2 种情况下构造函数接收字符串和两个整数；第 3 种情况下接收两个整数，第 1 个整数表示当年中的第几天。

6. 定义一个分数 Rational 类，进行分数的运算。用整数变量表示类的私有实例变量分子和分母。定义构造函数对所声明的对象初始化，并将分数存放成简化形式。例如对于 2/4 应存放成 1/2，即分子为 1 和分母为 2 的形式。不提供初始化值时提供一个默认的构造函数。定义完成下列功能的方法：

- 两个 Rational 值相加。
- 两个 Rational 值相减。
- 两个 Rational 值相乘。
- 两个 Rational 值相除。
- 按 a/b 形式输出 Rational 值，a 为分子，b 为分母。
- 按浮点数形式输出 Rational 值。

最后在另一个类中测试这个类。

第8章 继承

继承是面向对象编程中的主要特性之一。简单地说，继承是使用已有类创建新类的过程。创建新类时一方面吸收已有类的成员；另一方面还能增加新成员，并能修改已有类提供的服务。利用继承机制不仅可提高软件模块的复用和扩展，同时还可提高开发效率。

8.1 基类与派生类

8.1.1 protected 成员

在第 3 章介绍过，类成员访问修饰符除公有的（public）、私有的（private）与内部的（internal）以外，还有保护的（protected）。由 protected 修饰的成员称为保护成员。保护成员不能在类外被类对象访问，这一点与私有成员类似。保护成员对类的用户而言是私有的，但保护成员可以被派生类的方法与属性引用，这一点与私有成员是不同的。

基类的私有成员继承到派生类中，但派生类的方法与属性都不能直接访问。而保护成员一方面像私有成员一样不能被类外的对象访问，另一方面又能被派生类的方法与属性访问。因此，一旦类中声明了保护成员，就意味着该类可能要作为基类，派生类中要访问这些成员。

所有非私有基类成员在派生类中保持原访问修饰符（即基类的公有和保护成员在派生类中仍然是公有的和保护的）。

派生类可直接使用成员名访问从基类继承的公有和保护成员，但派生类方法覆盖（new、override）基类方法时，使用 base.方法名()的形式访问基类的方法。

8.1.2 基类与派生类的关系

在现实生活中，许多事物之间存在着或多或少的联系，有时一类事物具有另一类事物的全部特点，同时还具有自身的特点。例如在一个公司中，雇员是公司聘用的工作人员，经理是管理公司的一种特殊雇员，这类雇员拥有普通雇员的所有特征，同时还能得到公司发给的特殊津贴。我们使用两个类分别描述雇员和经理两类人员，并假设雇员类（Employee）包含姓名、工作部门和基本工资三个域（字段）及相关的属性和方法。经理类（Manager）包含姓名、工作部门、基本工资和特殊津贴四个域（字段）及相关的属性和方法。下面分别给出 Employee 和 Manager 的类代码声明。

例 8-1 阅读理解，提出改进方法。

```
//雇员类声明
class Employee                          //雇员类
{
    private string name;                //姓名
    private double salary;              //工资
    private string depart;              //工作部门
    public void Register(string name,double salary,string depart)          //注册方法
    {
        this.name=name;
        this.salary=salary;
        this.depart=depart;
    }
    public string Name                  //姓名属性
    {
        get
        {
         return name;
        }
        set
        {
            name=value;
        }
    }
    public double Salary                //工资属性
    {
        get
        {
            return salary;
        }
        set
        {
            salary=value;
        }
    }
    public string Depart                //部门属性
    {
        get
        {
            return depart;
        }
        set
        {
```

```
                depart= value;
            }
        }
        public void Show()                      //输出方法
        {
            Console.WriteLine("姓名：{0}\n工资：{1}\n部门：{2}",name,salary,depart);
        }
}
```

//经理类声明

```
class Manager                                   //经理类
{
        private string name;                    //姓名
        private double salary;                  //工资
        private string depart;                  //工作部门
        private double special;                 //特殊津贴
        public Register (string name,double salary,string depart,double
              special)                          //注册方法
        {
            this.name= name;
            this.salary= salary;
            this.depart= depart;
            this.special= special;
        }
        public string Name                      //姓名属性
        {
            get
            {
                return name;
            }
            set
            {
                name= value;
            }
        }
        public double Salary                    //工资属性
        {
            get
            {
                return salary;
            }
            set
            {
                salary= value;
```

```
            }
    }
    public string Depart                    //部门属性
    {
        get
        {
            return depart;
        }
        set
        {
            depart=value;
        }
    }
    public double Special                   //特殊津贴属性
    {
        get
        {
            return special;
        }
        set
        {
            special=value;
        }
    }
    public void Show()                      //输出方法
    {
        Console.WriteLine("姓名:{0}\n工资:{1}\n部门:{2}\n津贴:{3}",
                name,salary,depart,special);
    }
}
```

```
//主方法声明
static void Main(string[] args)
{
    Employee e1=new Employee();                   //创建雇员对象
    e1.Register("张三",4500,"技术科");              //注册雇员基本信息
    Console.WriteLine("雇员的初始信息:");
    e1.Show();
    e1.Salary+=300;                               //雇员加薪
    Console.WriteLine("工资增加300元后,工资总额:{0}",e1.Salary);
    Manager m1=new Manager();                     //创建经理对象
    m1.Register("陈鹏",5000,"技术科",200);          //注册经理信息
    Console.WriteLine("\n经理的初始信息:");
    m1.Show();
```

```
    m1.Salary+ = 400;                              //经理加薪
    m1.Special+ = 50;                              //提高经理的津贴
    Console.WriteLine("工资增加 400 元,津贴增加 50 元后,工资总额: 0}",
            m1.Salary+ m1.Special);
    Console.Readkey();
}
```

程序执行结果如下：

```
雇员的初始信息:
姓名: 张三
工资: 4500
部门: 技术科
工资增加300元后, 工资总额: 4800

经理的初始信息:
姓名: 陈鹏
工资: 5000
部门: 技术科
津贴: 200
工资增加400元,津贴增加50元后, 工资总额: 5650;
```

比较以上两个类，我们发现 Manager 类中大部分代码与 Employee 类相同。在 Manager 类中只是增加和修改了很少的代码，如新增特殊津贴字段 special（第 6 行），新增 Special 属性（47～57 行），同时对 Register 方法（7～13 行）的第 7 行和第 12 行进行了修改。能否利用 Employee 类作为基础，花少量的时间，创建满足需要的 Manager 类呢？答案是肯定的。在 C＃中，通过继承机制能轻松地解决这一问题。

在 C＃中，利用继承机制可以从已有类创建新类。一个新类从已有类中获取其特性，这种现象称为类的继承。换句话说，从已有类创建一个新的类，称为类的派生。在这里，已有类称为基类或父类，新建类称为派生类或子类。

一个派生类能够继承已有类中的几乎全部成员（构造方法和析构方法除外）。一般情况下，不必重复编写派生类中继承的成员方法。当然，在派生类中也可以根据需要增加新成员，并对继承的方法进行扩充和修改。通常，派生类比基类更加具体，它代表一组外延较小的对象，而基类则是派生类的抽象。

在 C＃中，借助“:”构建派生类。在构建派生类的过程中，还需制定基类与派生类各自的特性、访问修饰符的选择以及在派生类中对基类成员做何处理。

派生类声明的一般格式：

```
类修饰符 class 派生类名:基类名
{
    新增派生类成员
}
```

其中，基类名必须是已有类的名称，默认的基类名为 object（System. Object），可省略。派生类名是新创建的类类型的名字。

例 8-2　对例 8-1 的改进。利用继承机制，从雇员类（Employee）派生出经理类（Manager）。

该例使用例 8-1 中 Employee 类作为基类。派生类 Manager 声明如下：

```
class Manager : Employee                        //派生类-Manager
```

```
02: {
03:     private double special;                  //新增域-特殊津贴
04:     public void Register(string name,double salary,string depart,double
            special)
05:     {
06:         base.Register(name,salary,depart);   //访问基类的 Register 方法
07:         this.special=special;
08:     }
09:     public double Special                    //新增特殊津贴属性
10:     {
11:         get
12:         {
13:             return special;
14:         }
15:         set
16:         {
17:             special=value;
18:         }
19:     }
20:     public new void Show()                   //覆盖基类方法 Show
21:     {
22:         base.Show();                         //访问基类的 Show 方法
23:         Console.WriteLine("\n津贴：{0}",special);
24:     }
25: }
```

很明显，派生类 Manager 的代码较前例大大缩短。派生类 Manager 在继承 Employee 类全部成员的基础上，新增一个特殊津贴域 special，并对继承的部分方法进行了扩充和修改。这种编程风格是面向对象语言所倡导的，应该很好掌握。

实际上，在派生类 Manager 中同样包含四个域，除了该类的第 3 行声明 special 域之外，其余三个域名 name、salary 和 depart 都来自基类 Employee。由于基类中的域名为私有(private)特性，所以在子类 Manager 中不能直接访问，但可以通过公有(public)属性(09～19 行)和公有方法(20～24 行)对其私有域进行操作。另外，基类中的公有属性、公有方法被派生类 Manager 继承后，仍然保持其公有特性，且在派生类中都可以直接访问。

- base 关键字：在派生类中使用 base 关键字可以指代当前类的父类，但只限于在构造方法、实例方法和实例属性中使用。如代码中第 6 行使用 base. Register 访问基类的实例方法 Register，而第 22 行使用 base. Show 访问基类中的实例方法 Show。
- new 关键字：为了能在派生类中以“覆盖方式”声明与基类中同名的方法(当然新方法具有不同的处理功能)，可用 new 关键字对新方法进行修饰说明(第 20 行)。其中“覆盖”的两个方法应满足：方法名相同；方法中的参数个数相等以及参数类型相同。这时，在派生类中直接使用方法名是指调用派生类中定义的新方法，但使用 base. 方法名(第 22 行)将调用基类中的同名方法。如果不使用 new 进行修

饰说明，编译时将提示警告信息“请使用关键字 new”。另外，如果将派生类 Manager 的方法 Show()声明为：

```
20: public new void Show()                //覆盖基类方法 Show()
21: {
22:     Console.WriteLine("姓名：{0}\n工资：{1}\n部门：{2}",name,salary,depart);
23:     Console.WriteLine("\n津贴：{0}",special);
24: }
```

编译时将指出 22 行错误。其原因是，在派生类的方法 Show 中直接访问基类的私有域名 name、salary 和 depart 是非法的。如果希望在派生类中能直接访问基类的私有成员，又不允许在类之外访问，可以将基类中的私有成员（private）声明修改为保护成员（protected）。例如，将例 8-1 类 Employee 中 03～05 行修改为保护成员声明。代码如下：

```
01: class Employee                        //雇员类
02: {
03:     protected string name;            //姓名
04:     protected double salary;          //工资
05:     protected string depart;          //工作部门
        …                                 //以下代码不变
49: }
```

这时，派生类 Manager 的第 22 行输出将是合法的。

在 C# 中，一个类只能从一个直接基类派生，即 C# 仅支持单继承。而派生类又可作为其他派生类的基类，因此派生的多个相关联类将构成一个类层次。类层次中的最高层是 object 类（一个通用基类，该类提供一些公共接口），它是任何类的直接或间接基类。继承具有传递性，如果类 A 派生出类 B，类 B 派生出类 C，则类 C 会继承类 B 和类 A 中声明的所有成员。

8.2 派生类的构造函数

在 C# 中，基类的构造方法和析构方法均不能被继承。实例化派生类对象就是按照对象的类层次从低到高逐层调用其直接基类的构造方法，即在执行本层派生类构造方法体之前，先调用直接基类的构造方法……直到无直接基类的最高层类（object 类）。这时最先执行 object 类的构造方法体……最后执行最底层类的构造方法体。object 类是所有类的根，它有一个默认的构造方法。基类的构造方法总是为其派生类对象继承的基类实例变量初始化，使用 base 关键字调用基类的构造方法。例如：一个派生类的层次结构如图 8-1 所示。

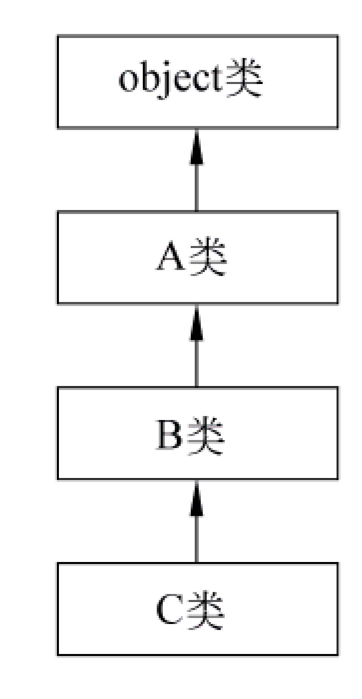


图 8-1 一个派生类的层次结构

在图 8-1 中，各类的继承关系如下：C 的直接基类是 B，B 的直接基类是 A，A 的直接基类是根类 object。

当实例化一个 C 类对象时，构造方法的执行过程是：先访问 C 类的构造方法；在执行 C 类构造方法体之前，先访问直接基类 B 的构造方法；在执行 B 类构造方法体之前，先访问直接基类 A 的构造方法；在执行 A 类构造方法体之前，先访问直接基类 object 的构造方法，这时已到达最高层-根类。接下来，依次执行 object 类的构造方法体、A 类的构造方法体、B 类的构造方法体，最后执行 C 类的构造方法体。到此一个 C 类对象创建完毕。

派生类构造方法的声明格式：

```
public 派生类名(参数总表):base(参数表)
{
    派生类中新增成员的初始化
}
```

其中：base(参数表)为显式访问基类中带有参数的构造方法。

说明：对于基类的默认构造方法或不带参数的构造方法，系统在实例化时会自动访问(隐式访问)，在这种情况下，可以省略显式调用。

例 8-3 派生类的构造方法(无参构造方法的隐式访问)。

程序代码如下：

```
//派生类的构造方法
using System;
namespace CSHAP8_4
{
    class Program
    {
        public class A:object           //注：:object 可以省略
        {
            public A()                  //不带参数的构造方法
            {
                Console.WriteLine("constructing A");
            }
        }
        public class B:A                //派生类 B
        {
            public B()                  //不带参数的构造方法
            {
                Console.WriteLine("constructing B");
            }
        }
        public class C:B                //派生类 C
        {
            public C()                  //不带参数的构造方法
            {
                Console.WriteLine("constructing C");
            }
```

```
27:         }
28:         static void Main(string[] args)
29:         {
30:             Console.WriteLine("创建 A 对象 a:");
31:             A a=new A();
32:             Console.WriteLine("创建 B 对象 b:");
33:             B n=new B();
34:             Console.WriteLine("创建 C 对象 c:");
35:             C c=new C();
36:         }
37:     }
38: }
```

运行结果如下：

```
创建 A 对象 a:
constructing A
创建 B 对象 b:
constructing A
constructing B
创建 C 对象 c:
constructing A
constructing B
constructing C
```

程序分析：

object 类的构造方法是一个空方法，无任何输出显示。

第 31 行，创建 A 类对象 a，初始化时调用 2 个构造方法。先执行 object 构造方法，无任何输出；后执行 A 的构造方法体，输出信息“constructing A”。

第 33 行，创建 B 类对象 b，初始化时调用 3 个构造方法。执行 object；执行 A 的构造方法体；执行 B 的构造方法体。输出 2 行信息。

第 35 行，创建 C 类对象 c，初始化时调用 4 个构造方法。执行 object；执行 A 的构造方法体；执行 B 的构造方法体；执行 C 的构造方法体。输出 3 行信息。

例 8-4 派生类构造方法（带参构造方法的显式访问）。

```
using System;
namespace CSHAP8_5
{
    class Program
    {
        public class A
        {
            public A(int i)
            {
                Console.WriteLine("constructing A:"+ i);
```

```
            }
        }
        public class B : A
        {
            public B(int j):base(j)              //base(j)显式调用基类的构造方法
            {
                Console.WriteLine("constructing B:"+ j);
            }
        }
        public class C : B
        {
            public C(int k):base(k)
            {
                Console.WriteLine("constructing C:"+ k);
            }
        }
        static void Main(string[] args)
        {
            Console.WriteLine("创建 A对象 a:");
            A a=new A(1);
            Console.WriteLine("创建 B对象 b:");
            B n=new B(2);
            Console.WriteLine("创建 C对象 c:");
            C c=new C(3);
        }
    }
}
```

程序执行结果：

```
创建 A对象 a:
constructing A:1
创建 B对象 b:
constructing A:2
constructing B:2
创建 C对象 c:
constructing A:3
constructing B:3
constructing C:3
```

例 8-5 对例 8-2 的改进。继承机制下派生类构造方法的应用。

带有构造方法的雇员类 Employee 声明如下：

```
class Employee                              //雇员类
{
    protected string name;                  //姓名
```

```
protected double salary;                        //工资
protected string depart;                        //工作部门
public Employee(string name,double salary,string depart)                        //构造方法
{
    this.name= name;
    this.salary= salary;
    this.depart= depart;
}
public string Name                              //名字属性
{
    get
    {
        return name;
    }
    set
    {
        name= value;
    }
}
public double Salary                            //工资属性
{
    get
    {
        return salary;
    }
    set
    {
        salary= value;
    }
}
public string Depart                            //部门属性
{
    get
    {
        return depart;
    }
    set
    {
        depart= value;
    }
}
public void Show()                              //输出方法
{
    Console.WriteLine("姓名：{0}\n工资：{1}\n部门：{2}",name,salary,depart);
```

```
48:        }
49: }
```

其中,06～11 行为构造方法,为其三个域(字段)初始化。

需要说明的是,如果基类使用带参数的构造方法,那么派生类也必须声明带参数的构造方法。这时,派生类 Manager 代码声明如下:

```
01: class Manager:Employee                              //派生类-经理类
02: {
03:        private double special;                      //特殊津贴
04:        public Manager(string name,double salary,string depart,double
                 special):base (name,salary,depart)     //派生类的构造方法
05:        {
06:            this.special=special;
07:        }
08:        public double Special
09:        {
10:            get
11:            {
12:                return special;
13:            }
14:            set
15:            {
16:                special=value;
17:            }
18:        }
19:        public new void Show()                       //覆盖基类方法 Show()
20:        {
21:            base.Show();                             //调用基类方法 Show()
22:            Console.WriteLine("津贴:{0}",special);
23:        }
24: }
```

其中 04～07 行为派生类 Manager 的构造方法。在 04 行中,通过 base (name, salary,depart)访问基类 Employee 的构造方法,用于基类成员的初始化。

测试用主方法:

```
//主方法
static void Main(string[] args)
{
       Employee e1=new Employee("张三",4500,"技术科");            //创建雇员类对象并初始化
       Console.WriteLine("雇员的初始信息:");
       e1.Show();
       e1.Salary+=300;
       Console.WriteLine("工资增加 300 元后,工资总额:{0}",e1.Salary);
```

```
    Manager m1=new Manager("陈鹏",5000,"技术科",200);              //创建经理类对象并初始化
    Console.WriteLine("\n经理的初始信息:");
    m1.Show();
    m1.Salary+=400;
    m1.Special+=50;
    Console.WriteLine("工资增加 400元,津贴增加 50元后,工资总额:{0}",
          m1.Salary+m1.Special);
    Console.ReadKey();
}
```

测试结果:

```
雇员的初始信息:
姓名:张三
工资:4500元
部门:技术科
工资增加 300元后,工资总额:4800元

经理的初始信息:
姓名:陈鹏
工资:5000元
部门:技术科
工资增加 400元后,津贴增加 50元后,工资总额:5650元
```

8.3　object 类

object(System.Object 的别名)类是所有类的直接或间接基类,因此,我们可以把任何类型的值赋给 object 类型的变量。该类提供了 7 个公共接口,任何对象都可以使用。这些方法名称及说明见表 8-1。

表 8-1　方法名称及说明

名　　称	说　　明
Equals(Object)	确定指定的对象是否等于当前对象
Equals(Object, Object)	确定指定的对象实例是否被视为相等
Finalize	允许对象在垃圾回收器之前尝试释放资源并执行其他清理操作
GetHashCode	作为默认哈希函数
GetType	获取当前实例的 Type
MemberwiseClone	创建当前 Object 的浅表副本
ReferenceEquals	确定指定的 Object 实例是否是相同的实例
ToString	返回表示当前对象的字符串

1. Equals 方法

该方法有重载。
方法签名：

```
public virtual bool Equals(Object obj)
public static bool Equals(Object objA,Object objB)
```

作用：比较两个对象是否相等。相等时返回 true，否则返回 false。
默认情况下，对于引用类型，比较的是引用的地址；对于值类型，比较的是值。
例如：

```
int a=2,b=3;
int []c1={1,2,3},c2=c1;
c2[0]=12;
Console.WriteLine(Equals(a,b));          //值类型的比较
Console.WriteLine(Equals(c1,c2));        //引用类型的比较
```

输出结果：

```
False
True
```

2. Finalize 方法

方法签名：

```
protected virtual void Finalize()
```

作用：允许对象在垃圾回收器回收该对象之前尝试释放资源并执行其他清理操作。

3. GetHashCode 方法

方法签名：

```
public virtual int GetHashCode()
```

作用：用作特定类型的哈希函数。返回一个能够标识内存中指定对象的整数。

4. GetType()方法

方法签名：

```
public Type GetType()
```

作用：获取当前实例运行时的类型信息(Type 对象)。

5. MemberWiseClone()方法

方法签名：

```
protected Object MemberWiseClone()
```

作用：创建一个新的对象，它是当前对象成员的副本(当前对象的浅表副本)。

6. ReferenceEquals 方法

方法签名：

```
public static bool ReferenceEquals(Object objA,Object objB)
```

作用：判断两个 Object 实例是否是同一个实例。如果参数是值类型，则会装箱，比较的是装箱后的对象实例。

注意：如果 objA 是与 objB 相同的实例，或者如果二者都为空引用，则为 true；否则为 false。

7. ToString()方法

方法签名：

```
public virtual string ToString()
```

作用：返回一个代表当前对象的字符串。

注意：默认情况下返回的是该对象所属类型的全名称。

习题

1. 定义圆类 Circle，包含半径 r，属性 R 能判断半径 r 的合理性(r≥0)；还包括计算圆面积的方法 double Area()。从 Circle 类派生出圆柱体类 Cylinder 类，新增圆柱体的高 h，属性 H 能判断高 h 的合理性(h≥0)，新增计算圆柱体体积的方法 double Volume()。在主方法中，创建一个 Cylinder 对象，并输出该对象的底面半径、高以及体积。(要求：不使用构造方法，并且类中的域为私有，方法为公有。)

2. 使用带参数的构造方法完成习题 1。

3. 定义基类 Person，并由此类派生 Teacher 类和 Student 类。基类中包含姓名、性别和年龄三个域(字段)，教师类中增加工号、职称和工资三个域，学生类中增加学号、班级、专业和入学成绩四个域，并在以上各类中添加属性、构造方法和显示对象信息的方法 Show()。在主方法中创建教师和学生对象(并带有初始值)，分别输出各自对象的相关信息。(要求：所有的域为私有，方法为公有。)

4. 设计一个汽车类 Vehicle，包含的数据成员有车轮个数 wheels 和车重 weight。小车类 Car 是它的子类，其中包含载人数 passenger_load。卡车类 Truck 也是 Vehicle 的子类，包含载人数 passenger_load 和载重量 payload，每个类都有相关数据的输出方法。在主方法中创建 Car 和 Truck 对象，分别输出各自对象的相关信息。

5. 在几何图形类 Shape(自己设计属性和方法)的基础上，派生类椭圆类 Ellipse，其属性为圆心坐标及半长轴和半短轴的长度，并通过构造方法对这些属性初始化，通过成员

方法计算椭圆面积。

6. 编写一个存储艺术作品的程序。艺术作品分为三类：Painting、Music 和 Chamber，Chamber 是 Music 中的一类。要求如下：

(1) 定义基类 Art，包含作品的著者、作品标题、作品诞生年份及其作品所属分类。

(2) Painting 类要求显示画的宽和高等尺寸信息。

(3) Music 类要求显示能够代表其中内容的关键信息，例如："D Major"。

(4) Chamber 类要求显示其中包括的演奏人员的数目。

(基类与派生类中字段成员均为私有成员，并使用带参数的构造函数提供初值。)

7. 从点类(Point)中派生出一个线段类(Line)，计算线段的长度。其中，Point 中包含以下方法，如带可选参数的构造方法 Point(int x=0,int y=0)，输出坐标点的方法 ShowPoint 以及读写坐标 x、y 的属性方法。Line 类中新增表示线段终点的字段 EndPoint、计算线段距离的方法 Distance 以及输出线段起点和终点的方法 ShowLine。

8. 设计评选优秀教师和学生的程序，其类结构如图 8-2 所示。当输入一系列教师或学生信息后，将优秀学生及教师的姓名和职业列出来。

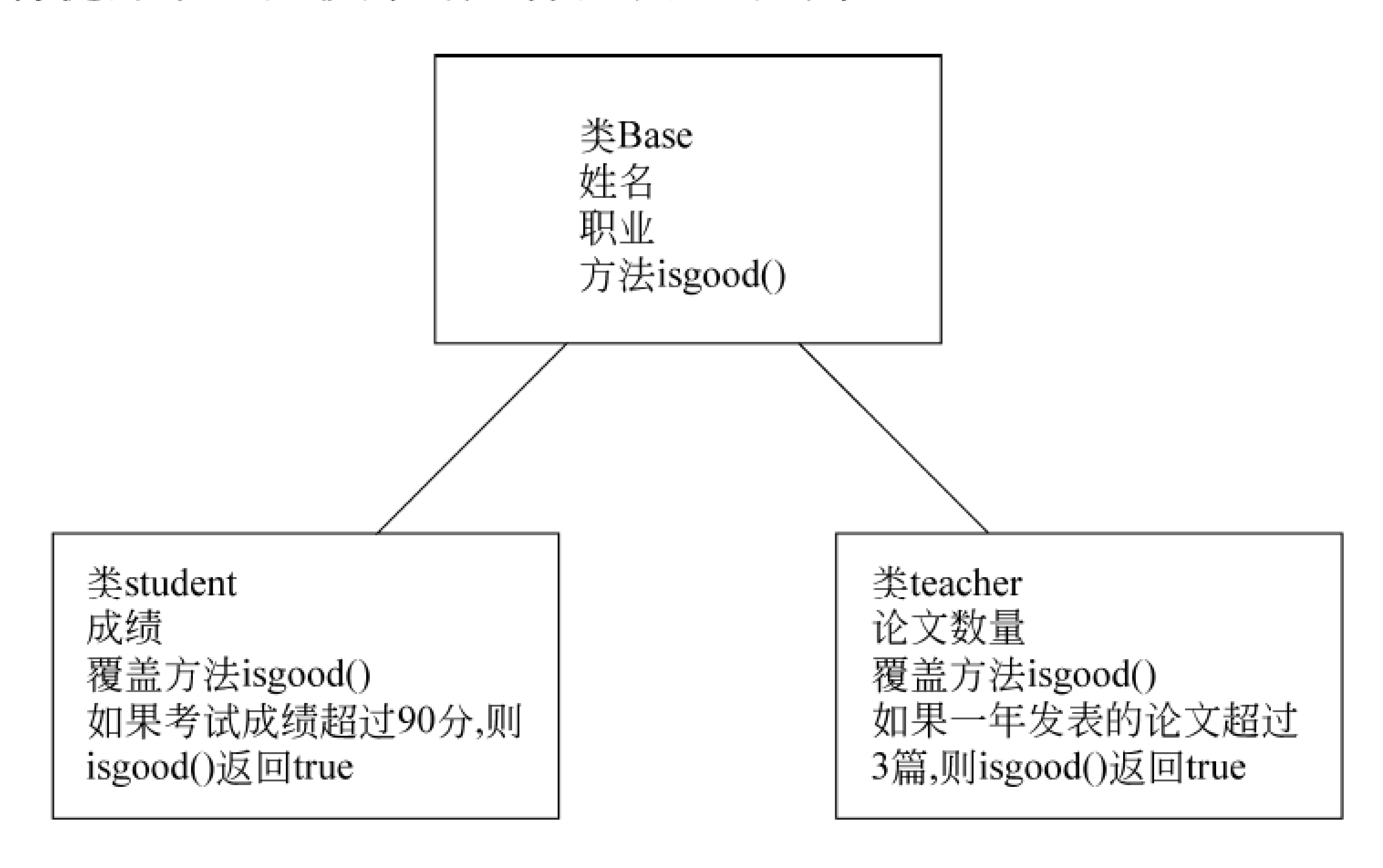
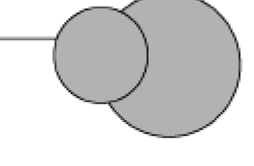


图 8-2 类结构

第9章 多态、接口和运算符重载



多态是面向对象程序设计理论中的重要概念。本章介绍多态、接口及运算符重载的概念以及它们的实现方式。

9.1 多态

多态是指同一个接口可以通过多种方法调用。换句话说，多态是指用一个相同的名字定义不同的成员函数，而这些函数执行不同的过程，即函数实现的功能不同。

多态是面向对象的程序设计的重要特性之一。使用多态，根据调用的对象类型的不同，同一方法名可以产生不同的动作。

本节先介绍多态的例子，然后用一个程序实例演示多态行为。

9.1.1 多态举例

先看第一个多态的例子，某个程序中要模拟几种动物的运动，三个类 Fish、Frog 和 Bird 是要研究的三种动物，每个类都继承基类 Animal，基类中包含 Move 方法方法和维护动物当前位置的 x、y 坐标，每个类实现方法 Move。

要模拟动物的运行，程序每秒钟向每个对象发送相同的消息 Move，但每种动物用不同的方式响应 Move 消息：Fish 游泳 3 英尺，Frog 跳 5 英尺，Bird 飞行 10 英尺。多态的关键就是每个对象知道如何响应相同的方法调用，并做出适合该类型对象的操作。同样的消息发到不同类型的对象时可以得到不同的结果，这就是多态。

如果对该基类 Animal 进行扩展，创建新的继承类 Tortoise，其响应的 Move 消息为爬行 1 英寸，只需要重写 Tortoise 类和实例化 Tortoise 对象的模拟部分，而处理一般 Animal 的模拟部分不需要改变。

再看一个四边形的例子。从一般的四边形 Quadrilateral 类定义派生类 Rectangle，则可以对 Quadrilateral 类的对象进行的任何操作也可以对 Rectangle 类的对象进行。同时这些操作也可以对 Quadrilateral 类的其他派生类进行，例如 Square、Parallelogram 和 Ttapezoid(不等边四边形)。在程序中通过基类变量调用方法时，发生多态，在执行时根据引用对象的类型调用这个方法的正确派生类版本。

在现实生活中也很容易列举出具有多态的例子。例如，不同型号的电视机，它们的内

部结构和电路组成是不完全相同的，但我们使用的电视机遥控器，其面板上的按键调节音量、换台等的作用却是类似的，即这些遥控器上都有一组相同或相似的控制键，从而可以使用相同的操作。遥控器就可以看成是这些不同型号电视机的统一接口，这一个接口控制不同的器件和电路。

在常用的程序设计语言(例如C++、C#语言)中也有多态的概念。例如，同一个运算符“－”，既可以完成两个整数的相减，也可以完成两个实数的相减；如果作为单目运算符，还可以实现取负操作。这是高级语言中固有的多态性。除了本身固有的多态性以外，还可以实现用户自定义的多态性。

利用多态性可以使得设计与实现的系统容易进行扩展，在增加新类时不必修改基本系统就可以将能够响应现有消息的新类型的对象添加到系统中，只需要重新编写实例化新对象的代码部分即可。

9.1.2 演示多态行为

在具有直接继承关系的两个类中，不同类中的对象引用自己类中的方法和属性，这是很平常的。例如基类的对象引用基类中的方法和属性，派生类的对象引用派生类的方法和属性。但是，在某些情况下，基类的对象也可以引用派生类的方法。

在C#中，派生类对象可以看作是基类对象，这样可以将基类当成派生类来处理，C#的编译器允许将基类引用赋予派生类变量，方法是显式地将基类引用转换为派生类类型，这样基类的对象就可以引用派生类的方法。

例 9-1 基类对象和派生类对象对方法的各种引用方式以及多态的演示。

本例演示三种方式引用类中的方法，分别是将基类引用赋予基类变量，派生类引用赋予派生类变量和将派生类引用赋予基类变量。

程序代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP9_1
{
    public class circle                    //定义基类圆形 circle
    {
        public double radius;
        public virtual void area()
        {
            Console.WriteLine("圆的面积={0}",Math.PI * radius * radius);
        }
        public circle(double r)
        {
            radius=r;
```

```
        }
    }
    public class globe : circle          //定义派生类球体 globe
    {
        public globe(double r)
            : base(r)
        {
        }
        public void volume()
        {
        Console.WriteLine("球体体积={0}",4.0/3.0*Math.PI*Math.Pow
                (radius,3));
        }
        public override void area()
        {
            Console.WriteLine("球体表面积={0}",4*Math.PI*radius*radius);
        }
    }
    public class cylinder : circle       //定义派生类圆柱体 cylinder
    {
        public double height;
        public cylinder(double r,double h)
            : base(r)
        {
            height=h;
        }
        public void volume()
        {
            Console.WriteLine("圆柱体体积={0}",Math.PI*radius*radius*height);
        }
        public override void area()
        {
            Console.WriteLine("圆柱体表面积={0}",2*Math.PI*radius*
                    radius+2*Math.PI*height);
            //圆柱体的表面积=侧面积+底面积*2
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            circle c1=new circle(3.5);
            Console.WriteLine("=========基类对象 c1 调用基类 circle 的方法
                    area():");
```

```
59:             c1.area();
60:             globe g1=new globe(4.0);
61:             Console.WriteLine("\n=========派生类对象 g1 调用派生类 globe 的方法 area()和
                    volume(): ");
62:             g1.area();
63:             g1.volume();
64:             cylinder cyl=new cylinder(3.0,4.0);
65:             Console.WriteLine("\n=========派生类对象 cyl 调用派生类 cylinder 的方法 area
                    ()和 volume(): ");
66:             cyl.area();
67:             cyl.volume();
68:             circle c2=g1;            //将派生类对象 g1 赋给基类变量 c2
69:             Console.WriteLine("\n=========基类对象 c2 调用派生类 globe 中的方法 area
                    (): ");
70:             c2.area();
71:             c2=cyl;                  //将另一个派生类对象 cyl 赋给基类变量 c2
72:             Console.WriteLine("\n=========基类对象 c2 调用派生类 cylinder 中的方法 area
                    (): ");
73:             c2.area();
74:         }
75:     }
76: }
```

程序的运行如下：

```
=========基类对象 c1 调用基类 circle 的方法 area():
圆的面积=38.484510006475
=========派生类对象 g1 调用派生类 globe 的方法 area()和 volume():
球体表面积=201.061929829747
球体体积=268.082573106329
=========派生类对象 cyl 调用派生类 cylinder 的方法 area()和 volume():
圆柱体表面积=81.6814089933346
圆柱体体积=113.097335529233
=========基类对象 c2 调用派生类 globe 中的方法 area():
球体表面积=201.061929829747
=========基类对象 c2 调用派生类 cylinder 中的方法 area():
圆柱体表面积=81.6814089933346
```

程序中第 8～19 行定义基类 circle，该类中有一个变量是半径 radius(第 10 行)，方法 area()(第 11～14 行)用于计算并显示圆的面积，构造函数(第 15～17 行)用来初始化对象。

第 20～34 行定义基类 circle 的派生类球体 globe，该类中有两个方法 volume()(第 26～29 行)和 area()(第 30～33 行)分别计算球体的体积和表面积。

第 35～52 行定义基类 circle 的派生类圆柱体 cylinder，该类中有一个变量是圆柱体

的高度 height（第 37 行），类中有两个方法 volume()（第 43～46 行）和 area()（第 47～51 行）分别计算圆柱体的体积和表面积。

第 53～75 用来验证和调用各个类中的方法。

第 57 行创建 circle 类的对象并且赋给了变量 c1。第 59 行用 c1. area()方式调用了 circle 类中的方法 area()来计算圆的面积。

第 60 行创建 globe 类的对象并且赋给了变量 g1。第 62、63 行分别调用了 globe 类中的两个方法 area()和 volume()计算球体的表面积和体积。

第 64 行创建 cylinder 类的对象并且赋给了变量 cy1。第 66、67 行分别调用了 cylinder 类中的两个方法 area()和 volume()计算球体的表面积和体积。

以上几个调用都是正常的调用即基类对象调用基类方法，派生类对象调用派生类方法。

第 68 行定义了基类对象 c2 并且赋值为派生类 globe 的引用。第 70 行中的方法调用 c2. area()；调用的是派生类 globe 中的方法。

第 71 行将基类对象 c2 赋值为派生类 cylinder 的引用，第 73 行的方法调用 c2. area()；调用的是派生类 cylinder 中的方法。

可以看出，使用基类变量的同样形式的方法调用 c2. area()；（第 70、73 行），具体调用哪个类中的方法取决于为这个基类变量赋值的引用类型。

如果定义一个基类的数组，数组中的每个元素也可以赋予不同的引用类型。例如，如果在本题 Main()的结尾添加下面的语句：

```
circle[] c=new circle [3];
c[0]=c1;
c[1]=g1;
c[2]=cy1;
c[0].area();
c[1].area();
c[2].area();
```

则这些语句的调用结果如下：

```
圆的面积=38.484510006475
球体表面积=201.061929829747
圆柱体表面积=81.6814089933346
```

同样的引用格式得到了不同的调用结果，这就是多态性的一种体现。

9.1.3　抽象类和方法

前面的例题中，在定义了一个类后，通常都要创建属于该类的对象，即类的实例化。也可以定义不实例化为任何对象的类，这种类称为抽象类。可以实例化对象的类则称为具体类。抽象类是要作为基类被其他类继承的，所以抽象类也称为抽象基类。

定义抽象类的唯一目的是为其他类提供合适的基类，其他类可以从它这里继承和(或)实现接口，从而共享相同的设计。

1. 声明抽象类

抽象类和抽象方法的声明都要使用关键字 abstract。声明抽象类的格式如下：

```
public abstract class 抽象类名
{
    声明类中的各个成员
}
```

例如，下面定义了一个几何形状的抽象类：

```
public abstract class shape
{
    protected double radius;
    public shape(double r) { radius=r; }
    public abstract double cubage();
}
```

在 shape 类中，有一个属性 radius、一个构造函数 shape()和一个抽象方法 cubage()。

一个类中包含有抽象方法时，该类必须定义为抽象类，抽象类 shape 是不能实例化的，如果在程序中出现下面的语句：

```
shape s=new shape(5.0);
```

在编译该程序时，将会出现"无法创建抽象类 shape 的实例"的错误。

2. 声明抽象方法和抽象属性

抽象类中通常包含一个或几个抽象方法。抽象方法是指在基类的定义中，不包含任何实现代码的方法，也就是不具有任何功能的方法，这种方法的功能将在派生类中实现。抽象方法在声明时使用 abstract 关键字进行修饰，其声明使用下面的格式：

```
[访问修饰符] abstract 返回值类型  方法名([参数列表]);
```

由于是抽象方法，因此只有声明部分，没有实现部分。

例如，上面定义的 shape 类中对抽象方法 cubage()的声明如下：

```
public abstract double cubage();
```

抽象类中也可以有抽象属性，在定义时也使用 abstract 关键字进行修饰。抽象属性不提供属性访问器的实现，访问器的实现也由派生类完成。

定义抽象属性的格式如下：

```
public abstract 返回值类型  属性名
{
    get;
    set;
}
```

在抽象类中可以包含抽象方法和抽象属性，也可包含非抽象的成员。

3. 重载抽象方法

在抽象类中，由于抽象方法和抽象属性都没有提供具体的实现，在定义抽象类的派生类时，在派生类中必须重载基类的抽象方法和属性。如果派生类没有进行重载，则派生类也必须声明为抽象类。

重载抽象方法的格式如下：

```
public override 方法名([参数列表]) {  }
```

例 9-2　抽象类和抽象方法的使用。

本题中定义一个抽象类 shape，其中包含抽象方法；然后分别定义 3 个该类的派生类，在各个派生中分别实现抽象类中的抽象方法。程序代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP9_2
{
    public abstract class shape
    {
        protected double radius;
        public shape(double r) { radius= r; }
        public abstract double cubage();
    }
    public class globe : shape                    //定义派生类球体 globe
    {
        public globe(double r) : base(r) { }
        public override double cubage()           //重载抽象方法
        {
            return Math.PI * radius * radius * radius * 4.0/3;
        }
        public override string ToString()         //重载 ToString()方法
        {
            return string.Format("球体体积为: {0}",cubage());
        }
    }
    public class cone : shape                     //定义派生类圆锥体 cone
    {
        public double height;
        public cone(double r,double h) : base(r) { height= h; }
        public override double cubage()           //重载抽象方法
```

```
31:             {
32:                 return Math.PI * radius * radius * height/3;
33:             }
34:             public override string ToString()
35:             {
36:                 return string.Format("圆锥体体积为: {0}",cubage());
37:             }
38:         }
39:         public class cylinder : shape                //定义派生类圆柱体 cylinder
40:         {
41:             public double height;
42:             public cylinder(double r,double h) : base(r) { height=h; }
43:             public override double cubage()          //重载抽象方法
44:             {
45:                 return Math.PI * radius * radius * height;
46:             }
47:             public override string ToString()
48:             {
49:                 return string.Format("圆柱体体积为: {0}",cubage());
50:             }
51:         }
52:         class Program
53:         {
54:             static void Main(string[] args)
55:             {
56:                 globe g=new globe(5.0);
57:                 Console.WriteLine(g);
58:                 cone c=new cone(5.0,10.0);
59:                 Console.WriteLine(c);
60:                 cylinder cy=new cylinder(5.0,10.0);
61:                 Console.WriteLine(cy);
62:             }
63:         }
64: }
```

程序的运行结果如下:

```
球体体积为: 523.598775598299
圆锥体体积为: 261.799387799149
圆柱体体积为: 785.398163397448
```

程序中第 8～13 行定义了一个抽象类 shape。该类中定义了 3 个成员：一个是实体变量 radius(第 10 行)；一个是构造函数(第 11 行)，用于初始化对象；一个是抽象方法 cubage()，表示计算体积，因为在该类中不知道是什么类型的形状，无法计算体积，所以定义为抽象方法，具体的计算方法在派生类中实现。

第 14～25 行定义了抽象类 shape 的第 1 个派生类球体 globe。该类中有 3 个成员，构造函数(第 16 行)中调用了基类的构造函数 base(r)，用来向基类传递参数。第 17～20 行对基类的抽象方法 cubage()进行了实现，该方法返回球体的体积。

第 21～24 行重载了 ToString()方法，该方法产生一个包含用 cubage()计算出的结果的输出字符串。

第 26～38 行定义了抽象类 shape 的第 2 个派生类圆锥体 cone。该类中有 4 个成员，实例变量 height(第 28 行)表示圆锥体的高度，构造函数(第 29 行)中调用了基类的构造函数 base(r)，用来向基类传递参数同时给变量 height 提供值。第 30～33 行对基类的抽象方法 cubage()进行了实现，该方法返回圆锥体的体积。第 34～37 行重载了 ToString()方法，该方法用 cubage()计算出的体积产生一个输出字符串。

第 39～51 行定义了抽象类 shape 的第 3 个派生类圆柱体。该类中也有 4 个成员，这 4 个成员与圆锥体 cone 类中的成员是相似的。

Main()方法中分别定义了 3 个派生类的对象 g、c 和 cy(第 56、58、60 行)。第 57 行的输出语句 Console. WriteLine(g)；中，默认调用了类中的方法 ToString()，而 ToString()方法中调用了计算体积的方法 cubage()，因此，第 57 行的输出中包含球体的体积计算结果。

同样，第 59、61 行分别输出了圆锥体和圆柱体的体积。

9.1.4　案例研究：使用多态的工资系统

例 9-3　本例是一个综合案例，用抽象和多态的方法，根据 4 类不同员工类型分别进行工资的计算。各类员工工资计算方法如下：

- 固定工：每周工资一样，与工作时间长短无关，由 SalariedEmployee 类实现；
- 计时工：按时计酬，超过 40 小时算加班工资，由 HourlyEmplyee 类实现；
- 雇佣员工：按销售百分比例计算，由 CommissionEmployee 类实现；
- 底薪雇佣员工：在底薪之上增加销售百分比。在本期内，公司准备对底薪雇佣员工升薪 10%，由 BasePlusCommissionEmployee 类实现。

用抽象基类 Employee 表示员工，这个类派生出 SalariedEmployee、HourlyEmplyee 和 CommissionEmployee 类。CommissionEmployee 类派生出 BasePlusCommissionEmployee 类。这 4 个类之间的继承层次关系如图 9-1 所示。

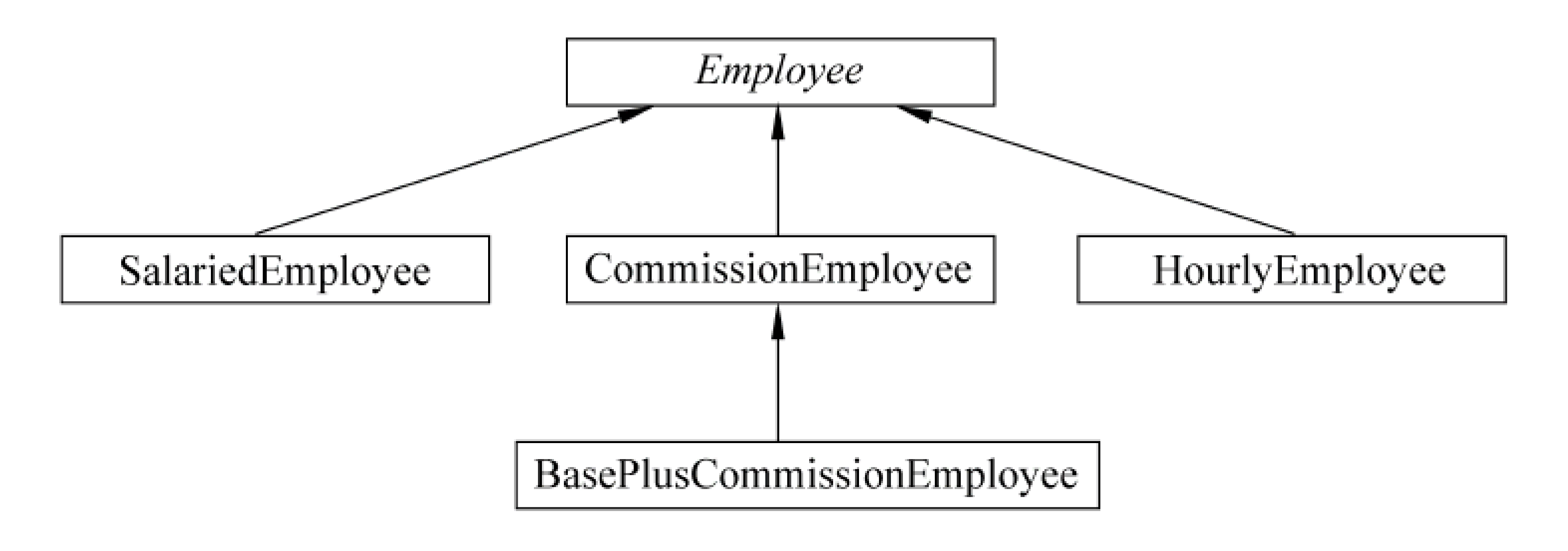


图 9- 1　Employee 及其派生类的层次关系

图 9-1 中用斜体表示的类名 Employee 为抽象类。

每个类中都包含 Earning 和 ToString 方法。由于每个类收入的计算方法不同，所以在基类中将 Earning()方法声明为 abstract，然后在每个派生类中进行相应的实现来覆盖该方法。

基类的 ToString()方法返回一个字符串，包含员工的名、姓和社会安全号。在每个派生中覆盖该方法，生成包含员工类型的字符串再加上其他信息。

每个类中这两个方法各自的结果见表 9-1。

表 9-1　Earning 和 ToString 方法在各个类中的结果

类　　名	Earning 方法	ToString 方法
Employee	abstract，没有实现部分	FirstName LastName Social Security Number：SSN
SalariedEmployee	weeklySalary	Salaried employee：FirstName LastName Social Security Number：SSN Weekly salary：weeklysalary
HourlyEmplyee	/If hours＜＝40 wage * hours //If hours＞40 40 * wage＋(hours－40) * wage * 1.5	Hourly employee：FirstName LastName Social Security Number：SSN Hours workey：wage Hours worked：hours
CommissionEmployee	commissionRate * grossSales	Commission employee：FirstName LastName Social Security Number：SSN Gross sales：grosssales Commission rate：commissionrate
BasePlusCommission-Employee	commissionRate * grossSales ＋baseSalary	Base salaried commission employee： FirstName LastName Social Security Number：SSN Gross sales：grosssales Commission rate：commissionrate Base salary：basesalary

以下分别定义各个类。

1. 创建抽象基类 Employee

定义抽象类 Employee 的代码如下：

```
//使用多态的工资系统案例第一部分:定义抽象基类 Employee
//定义员工的 3个属性、1个输出字符串方法 ToString()、1个抽象方法 Earning()
public abstract class Employee
{
    public string FirstName { get; private set; }   //员工的 FirstName
    public string LastName { get; private set;}     //员工的 LastName
    public string SocialSecurityNumber { get; private set; }
```

```
                                        //员工的社会安全号
8:      public Employee(string first,string last,string ssn)
                                        //带 3个参数的构造函数
9:      {
10:         FirstName= first;
11:         LastName= last;
12:         SocialSecurityNumber= ssn;
13:     }
14:     public override string ToString()          //定义输出字符串,包含员工的 3个属性
15:     {
16:         return string.Format("{0} {1}\nSocial security number: {2}",
            FirstName,LastName,SocialSecurityNumber);
17:     }
18:     public abstract decimal Earning();         //定义抽象方法,没有实现部分
19: }
```

Employee 类中,第 5～7 行定义了与员工有关的 3 个属性,即 FirstName、LastName 和 SocialSecurityNumber,基类中的属性在派生类中直接继承和使用。

第 8～13 行定义该类的构造函数,为简化程序,在构造函数中没有对社会安全号进行合理性的检验。

2. 创建具体派生类 SalariedEmployee

具体派生类 SalariedEmployee 扩展 Employee 类,实现 Earning()方法,代码如下:

```
//使用多态的工资系统案例第二部分:固定周工资
//定义 SalariedEmployee 类扩展 Employee 类
public class SalariedEmployee : Employee
{
    private decimal weeklySalary;           //周工资,变量名第 1个字母是小写
    public decimal WeeklySalary             //属性名第 1个字母是大写
    {
        get { return weeklySalary; }
        set { weeklySalary= ((value>=0)?value:0);}
    }
    public SalariedEmployee(string first,string last,string ssn,decimal salary)
        : base(first,last,ssn)              //带 4个参数的构造函数
    {
        weeklySalary= salary;
    }

    public override decimal Earning()
    {
        return weeklySalary ;
    }
```

```
21:     public override string ToString()
22:     {
23:         return string.Format("Salaried Employee:{0}\n{1}:{2:C}",
                base.ToString(),"Weekly salary",weeklySalary);
24:     }
25: }
```

程序的第 3 行继承基类 Employee,第 6～10 行的属性 WeeklySalary 用来操作实例变量 weeklySalary,其中第 9 行的 set 方法保证对 weeklySalary 赋予非负的值。

第 11～15 行的构造函数将员工的名、姓和社会安全号传入 Employee 类的构造函数(第 12 行),用初始化器(第 13～15 行)初始化派生类的专用实例变量 weeklySalary。

第 17～20 行 Earning()覆盖 Employee 类的抽象方法 Earning(),方法的具体实现是返回周工资 weeklySalary。

第 21～24 行的方法 ToString()覆盖 Employee 类的 ToString()方法。如果不覆盖,则该类会继承 Employee 类的 ToString()版本。派生类的 ToString()方法返回信息包括字符串"Salaried Employee"、基类 Employee 的特定信息(即名、姓和社会安全号)以及周工资 weeklySalary,其中的基类特定信息是调用基类 ToString()方法实现的,这是代码复用的体现。

3. 创建具体派生类 HourlyEmplyee

该类也是 Employee 类的扩展,代码如下。

```
//使用多态的工资系统案例第三部分:按时计酬
//定义 HourlyEmplyee 类扩展 Employee 类
public class HourlyEmplyee : Employee
{
    private decimal Wage;                    //小时工资
    private decimal Hours;                   //周工作小时
      public HourlyEmplyee (string first, string last, string ssn, decimal hourlyWage, decimal
      hoursWorked)
         : base(first,last,ssn)              //带 5 个参数的构造函数
    {
        Wage= hourlyWage;
        Hours= hoursWorked;
    }
    public decimal wage
    {
        get { return Wage; }
        set{Wage= (value >= 0) ? value : 0;}
    }
    public decimal hours
    {
        get { return Hours; }
```

```
21:         set { Hours= ((value>=0) && (value<=168))?value:0;}
22:     }
23:     public override decimal Earning()
24:     {
25:         if (Hours <=40)                    //周工作不超过 40 小时的计算
26:             return Wage * Hours;
27:         else
28:             return (40 * Wage)+ ((Hours - 40) * Wage * 1.5M);        //周工作超过 40 小时的计算
29:     }
30:     public override string ToString()
31:     {
32:         return string.Format("HourlyEmplyee Employee:{0}\n{1}:{2:C}{3}:{4:
        F2}", base.ToString(),"Hourly Wage",Wage,"hours Worked",Hours);
33:     }
34: }
```

该类定义了两个实例变量，分别是小时工资 Wage（第 5 行）和周工作小时 Hours（第 6 行）。

第 7～12 行是该类的构造函数，将名、姓和社会安全号传入基类的构造函数（第 8 行），初始化基类的专用实例变量（第 10、11 行）。

第 13～17 行是声明变量 Wage 的属性 wage，其中的 set 方法保证 Wage 为非负值，第 18～22 行声明变量 Hours 的属性 hours，其中的 set 方法保证 Hours 的值为 0～168，168 是一周 7 天的总小时数。

第 23～29 行的 Earning()方法计算 HourlyEmplyee 的收入，包括工作时间 40 小时以内和超过 40 小时的不同计算方法。

第 30～33 行派生类的方法 ToString()返回员工类型、员工特定信息和工资信息，也调用了基类的 ToString()方法，同样也是代码复用。

4. 创建具体派生类 HourlyEmplyee

该类的结构与前两个派生类相似，代码如下：

```
//使用多态的工资系统案例第四部分:销售百分比
//定义 HourlyEmplyee 类扩展 CommissionEmployee 类
public class CommissionEmployee : Employee
{
    private decimal GrossSales;                    //总销售额
    private decimal CommissionRate;                //佣金百分比
    public CommissionEmployee(string first,string last,string ssn,
    decimal sales,decimal rate)
        : base(first,last,ssn)                     //带 5 个参数的构造函数
    {
        GrossSales= sales;
        CommissionRate= rate;
```

```
12:     }
13:     public decimal commissionrage
14:     {
15:         get { return CommissionRate; }
16:         set { CommissionRate= (value > 0 && value< 1) ? value : 0; }
17:     }
18:     public decimal grosssales                        //操纵 GrossSales 的属性
19:     {
20:         get { return GrossSales; }
21:         set { GrossSales= (value >= 0)?value : 0;}
22:     }
23:     public override decimal Earning()
24:     {
25:          return CommissionRate * GrossSales ;
26:     }
27:     public override string ToString()
28:     {
29:         return string.Format("{0}:{1}\n{2}{3:C}\n{4}:{5:F2}","Commission
            Employee",base.ToString(),"Gross Sales:",GrossSales,"Commission
            Rate",CommissionRate);
30:     }
31: }
```

程序中的第 5、6 行定义了派生类中的两个实例变量总销售额 GrossSales 和佣金百分比 CommissionRate。

程序中同样包含构造函数(第 7～12 行)、操作两个实例变量的属性 commissionrage(第 13～18 行)和 grosssales(第 18～22 行)、实现基类的 Earning()方法(第 23～26 行)以及覆盖的 ToString()方法(第 27～30 行)。

5. 创建间接的具体派生类 BasePlusCommissionEmployee

本类的代码如下:

```
//使用多态的工资系统案例第五部分:底薪+销售百分比
//定义 BasePlusCommissionEmployee 类扩展 CommissionEmployee 类
public class BasePlusCommissionEmployee : CommissionEmployee
{
    public decimal BaseSalary;                    //每周基本工资
    public BasePlusCommissionEmployee(string first,string last,string
    ssn,decimal sales,decimal rate,decimal salary)
        : base(first,last,ssn,sales,rate)        //带 6个参数的构造函数
    {
        BaseSalary= salary;
    }
    public decimal baseSalary
```

```
12:     {
13:         get { return BaseSalary; }
14:         set { BaseSalary= (value >= 0)?value :0; }
15:     }
16:     public override decimal Earning()
17:     {
18:         return base.Earning()+BaseSalary ;
19:     }
20:     public override string ToString()
21:     {
22:         return string.Format ("Base-Salaried {0}:Base salary{1:C}",
            base.ToString(),BaseSalary);
23:     }
24: }
```

具体派生类 BasePlusCommissionEmployee 扩展 CommissionEmployee 类(第 3 行)，因此间接扩展 Employee 类。

该类自身的实例变量是每周基本工资 BaseSalary(第 5 行)，还有两个实例变量分别是总销售额 GrossSales 和佣金百分比 CommissionRate，都来自其基类 CommissionEmployee。

第 6～10 行是构造函数。

第 16～19 行是 Earning()方法的实现，计算 BasePlusCommissionEmployee 类的收入，该方法中调用了基类 CommissionEmployee 中的 Earning()方法(第 18 行中的“base.Earning()”)，这也是代码复用。

第 20～23 行是 ToString()方法的覆盖，用来生成 BasePlusCommissionEmployee 类的字符串表示，该字符串中包括调用 BasePlusCommissionEmployee 类的 ToString()方法产生的字符串和调用 Employee 类中的 ToString()方法产生的字符串和本类的 ToString()方法中的字符串，该方法跨越了 Employee 层次中的三层。

6. 多态处理、运算符 is 和向下转换

定义好各个类之后，接下来在 Program 类中测试 Employee 的各个层次。在程序中创建 4 个具体类的对象，然后通过每个对象自己的类型操纵这些对象，再用 Employee 数组进行多态操作，代码如下：

```
1: //使用多态的工资系统案例第六部分:多态处理
2: class Program
3: {
4:     static void Main(string[] args)
5:     {
6:         SalariedEmployee salariedEmployee=
               new SalariedEmployee("John","Smith","111-11-1111",800.00M);
7:         HourlyEmplyee hourlyEmplyee=
               new HourlyEmplyee("Karen","Price","222-22-2222",16.75M,40.0M);
```

```
            CommissionEmployee commissionEmployee=
                new CommissionEmployee("Sue","Jones","333-33-3333",10000.00M,.06M);
            BasePlusCommissionEmployee basePlusCommissionEmployee=new
                BasePlusCommissionEmployee("Bob","Lewis","444-44-4444",
            5000.00M,.04M,300.00M);
            Console.WriteLine("不同雇员的处理结果：");
            Console.WriteLine("{0}\nEarned:{1:C}\n",
                salariedEmployee,salariedEmployee.Earning());
            Console.WriteLine("{0}\nEarned:{1:C}\n",
                hourlyEmplyee,hourlyEmplyee.Earning());
            Console.WriteLine("{0}\nEarned:{1:C}\n",
                commissionEmployee,commissionEmployee.Earning ());
            Console.WriteLine("{0}\nEarned:{1:C}\n",
            basePlusCommissionEmployee,basePlusCommissionEmployee.Earning());
              Employee[] employees=new Employee[4];          //创建含有 4个元素的 Employee数组
            //使用 Employee的派生类初始化数组
            employees[0]=salariedEmployee;
            employees[1]=hourlyEmplyee ;
            employees[2]=commissionEmployee;
            employees[3]=basePlusCommissionEmployee;
            Console.WriteLine("不同雇员的处理结果：");
            foreach (var currentEmployee in employees)
            {
                Console.WriteLine(currentEmployee);     //调用 ToString()
                if (currentEmployee is BasePlusCommissionEmployee)
                {
                    BasePlusCommissionEmployee employee=
                        (BasePlusCommissionEmployee)currentEmployee;
                    employee.baseSalary *=1.10M;
                        Console.WriteLine("new base salary with 10%increase is:
                              {0:C}",employee.BaseSalary);
                }
                Console.WriteLine("Earned:{0:C}\n",currentEmployee.Earning());
            }
            for (int j=0; j<employees.Length; j++)
                Console.WriteLine("Employee{0} is a{1}",j,employees[j].GetType());
        }
}
```

程序的运行结果如下：

```
不同雇员的处理结果：
Salaried Employee:John Smith
Social security number: 111-11-1111
Weekly salary:￥800.00
```

```
Earned:￥800.00

HourlyEmplyee Employee:Karen Price
Social security number: 222-22-2222
Hourly Wage:￥16.75hours Worked:40.00
Earned:￥670.00

Commission Employee:Sue Jones
Social security number: 333-33-3333
Gross Sales:￥10,000.00
Commission Rate:0.06
Earned:￥600.00

Base-Salaried Commission Employee:Bob Lewis
Social security number: 444-44-4444
Gross Sales:￥5,000.00
Commission Rate:0.04:Base salary￥300.00
Earned:￥500.00

不同雇员的处理结果:
Salaried Employee:John Smith
Social security number: 111-11-1111
Weekly salary:￥800.00
Earned:￥800.00

HourlyEmplyee Employee:Karen Price
Social security number: 222-22-2222
Hourly Wage:￥16.75hours Worked:40.00
Earned:￥670.00

Commission Employee:Sue Jones
Social security number: 333-33-3333
Gross Sales:￥10,000.00
Commission Rate:0.06
Earned:￥600.00

Base-Salaried Commission Employee:Bob Lewis
Social security number: 444-44-4444
Gross Sales:￥5,000.00
Commission Rate:0.04:Base salary￥300.00
new base salary with 10% increase is:￥330.00
Earned:￥530.00

Employee 0 is a CSHARP9_3.SalariedEmployee
```

```
Employee 1 is a CSHARP9_3.HourlyEmplyee
Employee 2 is a CSHARP9_3.CommissionEmployee
Employee 3 is a CSHARP9_3.BasePlusCommissionEmployee
```

程序中，第 6～9 行创建了 4 个类的具体对象。分别是 salariedEmployee、hourlyEmplyee、commissionEmployee 和 basePlusCommissionEmployee。第 11～14 行输出每个对象的字符串表示和收入，其中每个对象的 ToString()方法在对象用格式输出为字符串时由 WriteLine()隐式地调用。

第 15 行声明了含有 4 个元素的 Employee 类的数组 employees。第 17～20 行将前面创建的 4 个不同类型的对象分别赋给数组中的每个元素，这些赋值都是允许的，因为派生类对象可以看作基类对象，所以不同的派生类变量都可以赋给基类变量。

第 22～32 行对数组中每个元素使用循环逐个处理，对 employees 的变量 currentEmployee 调用 ToString()方法和 Earning()方法，每次循环赋值不同的引用，输出显示实际上调用了每个类的适当方法。

这种在程序执行时根据 currentEmployee 所指的对象类型调用正确的 ToString()方法和 Earning()方法的过程称为动态绑定或后绑定，即在执行时而不是在编译时确定调用哪个 ToString()方法。

与动态绑定对应的是静态绑定，是指在编译时确定同名操作的具体操作对象，静态绑定通过函数重载实现。本章后面将要介绍的运算符重载就属于静态绑定。

循环体中的第 25～30 行对 BasePlusCommissionEmployee 对象进行了特殊的处理。在处理该类的对象时，将底薪增加 10%。

其中第 25 行中的“currentEmployee is BasePlusCommissionEmployee”中用 is 运算符确定对象 currentEmployee 是否为 BasePlusCommissionEmployee 类型，即采用了如下的格式：

```
对象名 is类名
```

第 27 行将 currentEmployee 从类层次结构中的类型 Employee 向下转换为类型 BasePlusCommissionEmployee，这是允许的，因为派生类和基类之间存在“是”关系。只有经过这个转换，才能对当前的 Employee 对象使用派生类 BasePlusCommissionEmployee 的属性 baseSalary，例如第 28 行中的 employee. baseSalary。

第 31 行调用 currentEmployee 的 Earning()方法。在这里是多态调用相应派生类对象的 Earning()方法，结果是取得 SalariedEmployee、HourlyEmplyee、CommissionEmployee 类各自计算的收入，结果与 11～13 行的结果相同。

第 31 行的多态从 BasePlusCommissionEmployee 计算的结果比第 14 行的结果(即收入)要多，因为底薪增加了 10%。

第 33、34 行显示每类员工的类型字符串，其中的 GetType()方法用来获得每个对象所属的类，该方法返回对象运行时的类，返回时隐式地调用 ToString()方法，ToString()方法返回对象所属的类名。

结合上面的例子，可以总结对基类变量和派生类变量赋值的基本要点如下：

(1) 将基类引用赋给基类变量是正确的。

(2) 将派生类引用赋给派生类变量是正确的。

(3) 将派生类引用赋给基类变量是安全的,因为派生类对象是基类对象,但这个引用只能引用基类成员。如果通过基类变量引用派生类中的特有成员,则会发生编译错误。

(4) 将基类引用赋给派生类变量是编译错误。如果一定要这样使用,应显式地将基类引用转换成派生类型。可以通过 is 运算符保证只对派生类对象进行转换。

9.2　sealed 方法和类

前面介绍的 abstract 修饰符所修饰的方法其内容缺少实现部分,这样在类声明中使用 abstract 修饰符的某个类只能是其他类的基类。标记为抽象或包含在抽象类中的成员必须通过从抽象类派生的类来实现,也就是说,抽象类必须有派生类。

与 abstract 相反的是,用 sealed 声明的基类不能有派生类,这样的类称为密封类。

当对一个类使用 sealed 修饰符时,其他类不能从该类继承。例如,在下面的代码中,类 B 从类 A 继承,但是任何类都不能从类 B 继承。

```
class A
{ }
sealed class B : A
{ }
```

如果使用下面的语句定义类 C:

```
class C : B
{ }
```

则程序在编译时会产生下面的出错信息:

```
无法从密封类型"XXXX.B"派生
```

其中的 XXXX 为对应的名字空间名称。

基类中的方法也可以使用 sealed 修饰,该方法称为密封方法。密封方法不能在派生类中覆盖,即用 sealed 声明的方法不能改变,因此所有的派生类使用相同的方法实现。

例 9-4　sealed 方法的使用。

程序中定义 3 个具有继承关系的类 X、Y 和 Z。每个类中有两个方法 F()和 F2(),其中有密封方法。程序如下:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP9_4
{
```

```
8:     class X
9:     {
10:        public virtual void F() { Console.WriteLine("X.F"); }
11:        public virtual void F2() { Console.WriteLine("X.F2"); }
12:    }
13:    class Y : X
14:    {
15:        public sealed override void F() { Console.WriteLine("Y.F"); }
16:        public override void F2() { Console.WriteLine("Y.F2"); }
17:    }
18:    class Z : Y
19:    {
20:        //protected override void F() { Console.WriteLine("C.F"); }
21:        //重载 F将会产生编译错误
22:        public override void F2() { Console.WriteLine("Z.F2"); }
23:        //重载 F2是允许的
24:    }
25:    class Program
26:    {
27:        static void Main(string[] args)
28:        {
29:            Z z=new Z();
30:            z.F();
31:            z.F2();
32:            Y y=new Y();
33:            y.F();
34:            y.F2();
35:        }
36:    }
37: }
```

程序的运行结果如下：

```
Y.F
Z.F2
Y.F
Y.F2
```

程序中第 8～12 行定义类 X,类中有两个方法 F()和 F2()。第 13～17 行定义 X 的派生类 Y,该类中也有两个方法 F()和 F2(),但 F()被声明为密封方法。Y 中的 F2()对 X 中的 F2()进行了重写。第 18～24 行定义了 Y 的派生类 Z,该类中有一个方法 F2(),对 Y 中的 F2()进行了重写。

第 27～35 行对定义的类进行了测试。第 29 行定义了 Z 类的对象 z。第 30 行调用了 Z 类的方法 F(),Z 类中的 F()是从 Y 类继承来的,所以输出“Y. F”。第 31 行调用了 Z 类

的方法 F2()，该方法是 Z 类中定义的，所以输出“Z. F2”。

第 32 行定义了 Y 类的对象 y，第 33、34 行对 Y 类的方法 F()和 F2()进行了调用，这两个方法都是 Y 类中重新定义的，所以分别输出“Y. F”和“Y. F2”。

如果将第 20 行的注释符“//”取消，程序在编译时该行就会产生下面的出错信息：

继承成员“CSHARP9_4. Y. F()”是密封的，无法进行重写

关于在派生类中的覆盖，有如下说明：

(1) 基类中的密封方法不能在派生类中覆盖。

(2) 声明为 private 的方法隐含 sealed，所以不能在派生类中覆盖。

(3) 声明为 static 的方法也隐含 sealed，所以也不能在派生类中覆盖。

(4) 同时声明为 override 和 sealed 的派生类方法可以覆盖基类的方法，但不能在继承层次的下层派生类中覆盖。

声明为 sealed 的类不能作为基类，该类中的所有方法隐含为 sealed。String 类就是一个 sealed 类。

9.3　创建和使用接口

一个接口定义一个协定。实现某接口的类或结构必须遵守该接口定义的协定。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

1. 接口的声明

接口的声明从 interface 开始，接口中的成员只能包含方法、属性、事件和索引器，也可以是这四个成员类型的任意组合。接口本身不提供它所定义的成员的具体实现，接口只指定实现该接口的类或结构必须提供的成员。

声明一个接口的格式如下：

```
[接口修饰符] interface 接口名 [: 基类接口名]
{
    接口的成员;
}
```

在上面的格式中：

- 接口的修饰符可以是 new、public、protected、internal 和 private。new 修饰符是在嵌套接口中唯一被允许存在的修饰符，它说明用相同的名称隐藏一个继承的成员，其他几个修饰符控制接口的访问能力。
- 接口名通常以字母“I”开始。
- 一个接口声明中可以声明零个或多个成员。接口的成员包括从基接口继承的成员和由接口本身声明的成员，所有接口成员默认都是 public，所以不能使用修饰符 abstract、public、protected、internal、private、virtual、override 或 static 来声明接口成员。接口成员之间不能重名。

例如，下面定义了一个名为 ImyPort 的接口，接口中有 4 个成员：

```
public interface ImyPort
{
    void display(string s);                     //方法成员
    int count{get;}                             //属性成员
    event stringList Changed;                   //事件成员
    string this[int index] { get; set; }        //索引成员
}
```

2. 接口的继承

接口也可以继承。一个接口可以从1到多个接口中继承,被继承的这些接口称为这个接口的基本接口,接口继承其基本接口中的所有成员。如果一个类或结构实现某接口,则它还隐式实现该接口的所有基接口。

接口的继承使每个接口可以扩展一个或多个其他接口,使其他类可以实现更完美的接口,这一点类似于类的继承性。

定义继承接口时,在接口名称后面用“:”加上被继承接口的名称,有多个被继承的接口时,接口名之间用“,”隔开。

例如,下面定义的接口D同时继承了接口IA、IB和IC:

```
interface ID:IA,IB,IC
{
}
```

上一章介绍过类的继承,它和接口的继承有如下区别:

- 类继承不仅说明继承,而且也实现继承,接口继承只说明继承,即只继承了父接口的成员方法说明,并没有继承父接口的实现。
- 在C#中类的继承只允许单继承,接口的继承允许多个继承,一个子接口可以有多个父接口。

3. 接口的实现

接口方法通常是通过类中的公有方法实现的。

例9-5 接口的继承与接口中方法的实现。

本例说明接口的定义、继承和接口的成员方法在继承类中的具体实现,程序如下:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    public interface IPortA
    {
```

```
10:         void DisplayStringA();
11:     }
12:     public interface IPortB:IPortA
13:     {
14:         void DisplayStringB();
15:     }
16:     public class C : IPortB
17:     {
18:         public void DisplayStringA()
19:         {
20:             Console.WriteLine("这是实现接口 IPortA 的结果");
21:         }
22:         public void DisplayStringB()
23:         {
24:             Console.WriteLine("这是实现接口 IPortB 的结果");
25:         }
26:     }
27:     class Program
28:         {
29:             static void Main()
30:             {
31:                 C c= new C();
32:                 c.DisplayStringA();
33:                 c.DisplayStringB();
34:             }
35:         }
36: }
```

程序的运行结果如下：

```
这是实现接口 IPortA 的结果
这是实现接口 IPortB 的结果
```

程序中第 8～15 行定义了两个接口 IPortA 和 IPortB。两个接口中各声明了一个方法分别是 DisplayStringA()和 DisplayStringB()，都没有实现部分。接口 IPortB 继承 IPortA，也就是继承了接口 IPortA 的方法，这样接口 IPortB 中就有两个未实现的方法。

第 16～26 行定义了类 C，类 C 继承了接口 IPortB，并且实现了从接口 IPortB 继承下来的两个方法(第 18～21、22～25 行)。

第 32、33 行分别调用了这两个方法。

接口常用于不同的类共享共同的方法，使不相关的对象可以多态处理，实现同一接口的类对象可以响应同一方法调用。

4. 显式实现接口的方法

在 C# 中，一个类中可以实现多个接口，例如下面对类 C 的声明：

```
public interface IPortA
{
    void DisplayString();
}
public interface IPortB
{
    void DisplayString();
}
public class C : IPortB,IPortA
{… …
}
```

类 C 继承的接口 IPort A 和接口 IPort B，因此在类 C 中要同时实现接口 A 和接口 B。

注意到两个接口中定义的方法名称相同，都是 DisplayString()，因此在类 C 继承的多个接口中存在同名的成员。为了区分是从哪个接口继承来的，可以采用显式实现接口的方法，显式实现方法的格式如下：

接口名称.成员名称

显式实现的成员不能带有任何修饰符。显式实现的成员不能通过类的实例进行调用，必须通过所属的接口来调用。这时采用的格式如下：

((接口名)类对象名).方法名

显式实现也可以用于实现两个接口分别声明具有相同名称的不同成员。例如属性和方法，对于同名的属性成员，可以采用下面的格式进行引用：

((接口名)类对象名).属性名

例 9-6 显式接口中方法的实现与调用。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    public interface IPortA
    {
        void DisplayString();
    }
    public interface IPortB
    {
        new void DisplayString();
    }
```

```
16:      public class C : IPortB,IPortA
17:      {
18:          void IPortA.DisplayString()        //采用格式:接口名称.成员名称
19:          {
20:              Console.WriteLine("这是实现接口 A的结果");
21:          }
22:          void IPortB.DisplayString()
23:          {
24:              Console.WriteLine("这是实现接口 B的结果");
25:          }
26:      }
27:      class Program
28:          {
29:              static void Main()
30:              {
31:                  C c=new C();
32:                  ((IPortA)c).DisplayString();   //采用格式 (接口名)类对象名).方法名
33:                  ((IPortB)c).DisplayString();
34:              }
35:          }
36: }
```

程序的运行结果如下：

```
这是实现接口 IPortA的结果
这是实现接口 IPortB的结果
```

程序中第 8～15 行定义了两个接口 IPortA 和 IPortB。这两个接口之间没有继承关系，两个接口中都有一个同名的方法 DisplayString()。

第 16～26 行定义了类 C，类 C 继承了两个接口 IPortA 和 IPortB，并且实现了两个接口中的同名方法。

类成员 IPortA. DisplayString() 只能通过 IPortA 接口调用，同样，类成员 IPortB. DisplayString() 只能通过 IPortB 接口调用。第 18～21 行通过 IPortA. DisplayString() 显式实现了接口 IPortA 的方法。第 22～25 行通过 IPortB. DisplayString() 显式实现了接口 IPortB 的方法，因此这两个同名的方法有了不同的实现。

第 32、33 行分别调用了实现不同接口的方法。

5. C# 中抽象类和接口的区别

接口含有多个不同的成员，其方法成员要在派生类中实现，所以接口和抽象类有下面的相似之处：

- 不能实例化。
- 包含未实现的方法声明。
- 派生类必须实现未实现的方法，抽象类是抽象方法，接口则是所有成员。

它们之间的区别如下：

抽象类具有下列特点：

- 抽象类是特殊的类，不能被实例化；除此以外，具有类的其他特性。
- 抽象类可以包括抽象方法，普通类则不能。抽象方法只能声明于抽象类中，且不包含任何实现，派生类必须覆盖它们。
- 抽象类可以派生自一个抽象类，可以覆盖基类的抽象方法也可以不覆盖。如果不覆盖，则其派生类必须覆盖它们。

接口具有下列特点：

- 接口可以包含方法、属性、索引器、事件 4 种成员，这些成员都被定义为公有的。
- 一个类可以直接继承多个接口，但只能直接继承一个类（包括抽象类）。
- 可以将接口看成只包含抽象成员并且所有成员都没有实现的抽象类。

在具体使用时，抽象类用于部分实现一个类，再由用户按需求对其进行不同的扩展和完善；接口只是定义了一个行为的规范或规定。

抽象类主要用于关系密切的对象，而接口适合为不相关的类提供通用功能。

抽象类主要用于设计大的功能单元，而接口用于设计小而简练的功能块。

9.4 运算符重载

本节介绍函数重载（方法重载）的一个特殊情况——运算符重载的概念和使用。

在 C# 中，对于预定义的运算符，其操作对象只能是基本的数据类型。例如，对于运算符“+”，如果有两个操作对象 x 和 y，不论它们是整型数还是实型数，都能实现 x+y，因为整型或实型是基本的类型。

对于下面定义的复数类 ComplexNumber：

```
class ComplexNumber
{
    public double real;
    public double imaginary;
    public ComplexNumber(double r,double i)
    {
        real=r;
        imaginary=i;
    }
}
```

如果定义了该类的两个对象 x 和 y：

```
ComplexNumber x=new ComplexNumber (1,2);
ComplexNumber y=new ComplexNumber (3,4);
```

现在要计算这两个复数的和，即计算表达式 x+y，那么该表达式在编译时会出错，因为这里的运算对象不是基本类型，而是类 ComplexNumber 的对象，无法实现用运算符“+”预

定义的功能。

为实现上述功能，需要自己编写程序来说明“+”用于类对象时，完成什么样的具体功能，也就是需要定义新的运算符函数即重载函数，这就是运算符的重载。显然，运算符的重载是对已有的运算符赋予多重含义，使同一个运算符用于不同类型数据时，产生不同类型的结果。

实现重载功能时，可以将重载函数声明为该类的成员函数。这时，先把指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的实参；然后，根据实参的类型确定要调用的函数。

将一个运算符重载为类的成员函数时，它的声明格式如下：

```
public static 返回值类型 operator 运算符 ([形参表列])
{
    函数体
}
```

在上面的格式中，各部分的含义如下：

(1) 返回值类型是重载的运算符函数返回值的类型，即运算结果的类型。

(2) operator 是定义运算符重载函数关键字。

(3) 运算符就是要重载的运算符本身的符号。

(4) 形参表列是重载运算符所需要的参数及类型。

例 9-7　将四则运算运算符重载为成员函数完成复数的四则运算。

两个复数四则运算的结果也是复数。设两个复数分别是 $a+bi$ 和 $c+di$，四则运算的规则如下：

- 加法法则：$(a+bi)+(c+di)=(a+c)+(b+d)i$
- 减法法则：$(a+bi)-(c+di)=(a-c)+(b-d)i$
- 乘法法则：$(a+bi)*(c+di)=(ac-bd)+(bc+ad)i$
- 除法法则：$(a+bi)/(c+di)=(ac+bd)/(c^2+d^2)+(bc-ad)/(c^2+d^2)i$

程序代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP9_5
{
    class ComplexNumber
    {
        public double real;
        public double imaginary;
        public ComplexNumber(double r,double i)
        {
```

```
            real= r;
            imaginary= i;
        }
        public string ShowString()
        {
            //设计输出字符串
            return Convert.ToString(real)+ (imaginary< 0?"- ":"+ ")+
            Convert.ToString (Math.Abs(imaginary))+ "i";
         }
        public static ComplexNumber operator+ (ComplexNumber x,
        ComplexNumber y)                                          //运算符"+ "重载为成员函数
        {
            //计算规则: (a+ bi)+ (c+ di)= (a+ c)+ (b+ d)i
            return new ComplexNumber (x.real+ y.real,x.imaginary+ y.imaginary);
        }
        public static ComplexNumber operator - (ComplexNumber x,
        ComplexNumber y)                                          //运算符"- "重载为成员函数
        {
            //计算规则: (a+ bi)- (c+ di)= (a- c)+ (b- d)i
            return new ComplexNumber(x.real - y.real,x.imaginary - y.imaginary);
        }
        public static ComplexNumber operator * (ComplexNumber x,ComplexNumber y)
        //运算符"* "重载为成员函数
        {
            double r,i;
            //计算规则: (a+ bi) * (c+ di)= (ac- bd)+ (bc+ ad)i
            r=x.real * y.real - x.imaginary * y.imaginary;
            i=x.real * y.imaginary+ y.real * x.imaginary;
            return new ComplexNumber (r,i);
        }
        public static ComplexNumber operator /(ComplexNumber x,
        ComplexNumber y)                                          //运算符"/"重载为成员函数
        {
            double r,i;
            //计算规则: (a+ bi)/(c+ di)= (ac+ bd)/(c^2+ d^2)+ (bc- ad)/(c^2+ d^2)i
            r= (x.real * y.real+ x.imaginary * y.imaginary)/(y.real *
              y.real+ y.imaginary * y.imaginary);
            i= (x.imaginary * y.real - x.real * y.imaginary)/(y.real *
              y.real+ y.imaginary * y.imaginary);
            return new ComplexNumber(r,i);
        }
        class Program
        {
            static void Main(string[] args)
```

```
52:                 {
53:                     ComplexNumber x,y,z;
54:                     double r,i;
55:                     Console.Write("请输入第一个复数的实部:");
56:                     r=Convert.ToDouble(Console.ReadLine());
57:                     Console.Write("请输入第一个复数的虚部:");
58:                     i=Convert.ToDouble(Console.ReadLine());
59:                     x=new ComplexNumber (r,i);
60:                     Console.Write("请输入第二个复数的实部:");
61:                     r=Convert.ToDouble(Console.ReadLine());
62:                     Console.Write("请输入第二个复数的虚部:");
63:                     i=Convert.ToDouble(Console.ReadLine());
64:                     y=new ComplexNumber (r,i);
65:                     z=x+y;
66:                     Console.WriteLine("复数相加:({0})+({1})=({2})",
                        x.ShowString(),y.ShowString(),z.ShowString());
67:                     z=x-y;
68:                     Console.WriteLine("复数相减:({0})-({1})=({2})",
                        x.ShowString(),y.ShowString(),z.ShowString());
69:                     z=x*y;
70:                     Console.WriteLine("复数相乘:({0})*({1})=({2})",
                        x.ShowString(),y.ShowString(),z.ShowString());
71:                     z=x/y;
72:                     Console.WriteLine("复数相除:({0})/({1})=({2})",
                        x.ShowString(),y.ShowString(),z.ShowString());
73:                 }
74:             }
75:         }
76: }
```

程序运行的结果如下:

```
请输入第一个复数的实部:2
请输入第一个复数的虚部:4
请输入第二个复数的实部:4
请输入第二个复数的虚部:-2
复数相加:(2+4i)+(4-2i)=(6+2i)
复数相减:(2+4i)-(4-2i)=(-2+6i)
复数相乘:(2+4i)*(4-2i)=(16+12i)
复数相除:(2+4i)/(4-2i)=(0+1i)
```

程序中的 ComplexNumber 类重载了四则运算符加(+)、减(-)、乘(*)和除"/",实现了对复数进行四则运算。

第17~21行定义方法 ShowString()用来设计输出字符串,产生形如3+4i的字符串。

第 22～26 行重载加法运算符，完成复数的加法运算，方法头部的关键字 operator＋用来指定重载运算符的运算符是“＋”。重载的双目运算符要用到两个操作数，这就是形参表列中的两个参数 ComplexNumber x 和 ComplexNumber y。方法的结果返回的是两个复数之和。在操作符重载中，返回值往往要用“new”创建一个新的 ComplexNumber 对象。

注意：该方法的修饰符有公用 public 和静态 static，这是重载运算符必需的。

第 27～31 行、32～40 行、41～48 行分别重载减法、乘法和除法运算。

整个程序完成了两个复数的四则运算。第 55～64 行提示分别输入 4 个实数，即两个复数的实部和虚部，然后用输入的实数创建了两个复数 x 和 y(第 59 行、第 64 行)。

第 65 行进行了两个复数的加法运算。如果不使用运算符“＋”的重载，则对于复数 x 和 y，表达式 x＋y 是没有意义的。通过重载函数可以完成复数加法运算。第 66 行显示的就是复数加法的结果。

第 67～72 行分别完成两个复数的减法、乘法和除法运算并显示计算的结果。

例 9-8 重载运算符“＋＋”，使其用于复数的运算，运算规则是实部和虚部分别加 1。

程序代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace CSHARP9_6
{
    class ComplexNumber
    {
        public double real;
        public double imaginary;
        public ComplexNumber(double r,double i)
        {
            real= r;
            imaginary= i;
        }
        public string ShowString()
        {
            return Convert.ToString(real)+ (imaginary< 0?"- ":"+ ")+
            Convert.ToString (Math.Abs(imaginary))+ "i";
        }
        public static ComplexNumber operator+ + (ComplexNumber x)
        {
            double r= x.real+ 1;
            double i= x.imaginary+ 1;
            return new ComplexNumber(r,i);
        }
```

```
26:         class Program
27:         {
28:             static void Main(string[] args)
29:             {
30:                 ComplexNumber x,y,z;
31:                 double r,i;
32:                 ComplexNumber k=new ComplexNumber (1,2);
33:                 Console.Write("请输入复数的实部:");
34:                 r=Convert.ToDouble(Console.ReadLine());
35:                 Console.Write("请输入复数的虚部:");
36:                 i=Convert.ToDouble(Console.ReadLine());
37:                 x=new ComplexNumber (r,i);
38:                 y=++x;
39:                 Console.WriteLine("x={0}",x.ShowString());
40:                 Console.WriteLine("y={0}",y.ShowString());
41:                 z=x++;
42:                 Console.WriteLine("x={0}",x.ShowString());
43:                 Console.WriteLine("z={0}",z.ShowString());
44:             }
45:         }
46:     }
47: }
```

程序的运行结果如下：

```
请输入复数的实部:1
请输入复数的虚部:2
x=2+3i
y=2+3i
x=3+4i
z=2+3i
```

第 20～25 行重载自增运算符“＋＋”，分别对实部和虚部加 1。重载的单目运算符要用到一个操作数，这就是形参表列中的参数 ComplexNumber x。

第 32～36 行提示分别输入 2 个实数，即复数的实部和虚部，然后用输入的实数创建了复数 x(第 37 行)。

第 38 行进行了复数的前置加 1 运算：y＝＋＋x;，变量 y 保存运算的结果，第 39、40 行分别输出复数 x 和 y。从输出结果可以算出，变量 x 的值进行了加 1 运算，由于进行了前置加 1 运算，变量 y 的值是 x 加 1 以后的值。

第 41 行进行了复数的后置加 1 运算：z＝x＋＋;，变量 z 保存运算的结果，第 42、43 行分别输出复数 x 和 z，从输出结果可以看出，变量 x 的值进行了加 1 运算。由于进行了后置加 1 运算，变量 z 的值是 x 加 1 以前的值。

在 C＃中，可以重载的运算符如下：

单目运算符：＋、－、!、～、＋＋、－－。

双目运算符：＋、－、＊、/、％、＆、|、^、＜＜、＞＞、＝＝、!＝、＞、＜、＞＝、＜＝。

对运算符进行重载时应注意下面的问题：

(1) 只能重载 C# 中已有的运算符，不能通过重载定义新的运算符。

(2) 运算符的重载不会改变原运算符的优先级和结合性。

(3) 运算符的重载不会改变使用运算符的语法规则和参数个数，即单目运算符只能重载为单目运算符，双目运算符只能重载为双目运算符。

(4) 重载后的功能应该与原有的功能类似。

习题

1. 多态如何实现“通用编程”而不是“特定编程”？“通用编程”的好处是什么？

2. 派生类可以从基类继承“接口”或“实现”，继承接口的继承层次和继承实现的继承层次有什么不同？

3. 什么是抽象方法？什么是抽象类？

4. 接口和抽象类有什么异同？什么情况下使用接口？什么情况下使用抽象类？

5. 修改本章的例题 9-3，在 Employee 类中添加新的实例变量 birthDate。用 Date 类表示员工的生日，假设工资每月处理一次。创建 Employee 数组，存储不同类型员工对象的引用。在循环中计算每个员工的工资。如果本月是员工的生日，则工资增加 100 元。

6. 修改本章的例题 9-3，增加 Employee 类的另一个派生类 PieceWorker，表示计件工。PieceWorke 类具有专用的实例变量 wage 和 pieces 分别表示每件的工资和生产的件数。在 PieceWorker 类中提供 Earning 方法的具体实现，计算员工的收入。计算方法是将件数乘以每件的工资。创建一个 Employee 数组，存储新的类层次中每个具体类对象的引用，显示每个员工的字符串表示和收入。

第10章 异常处理

一个程序可能按我们的意愿正常运行到终止，例如使用 return 语句或调用 exit()函数；也可能因为出现某种异常而意外终止。

这里的异常就是程序在执行期间发生的问题。例如程序执行时遇到除数为 0 的除法运算，对负数进行开平方，数组的下标越界等，这时系统将产生中断，从而导致正在执行的程序提前终止。

异常处理机制正是 C# 中用于管理程序运行期间错误的一种结构化方法。异常处理机制将程序中的正常处理代码与异常处理代码明显区别开来，提高了程序的可读性。

异常处理机制的基本思想是将异常的检测与处理分离。当在一个函数体中检测到异常条件存在，但无法确定相应的处理方法时，将引发一个异常，并由函数的直接或间接调用者检测并处理这个异常。

本章介绍异常处理的概念、.NET 的异常处理类层次、异常处理的程序结构及常用的异常处理语句。

10.1 异常处理的例子

1. 异常现象的抛出

下面先看一个没有进行异常处理的控制台程序出现错误的情况。

例 10-1 没有进行异常处理的程序。

以下是计算两个整数整除的程序：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP10_1
{
    class Program
    {
        static void Main(string[] args)
```

```
11:            {
12:                Console.WriteLine("请输入第一个整数：");
13:                int number_1=Convert.ToInt32(Console.ReadLine());
14:                Console.WriteLine("请输入第二个整数：");
15:                int number_2=Convert.ToInt32(Console.ReadLine());
16:                int result=number_1/number_2;
17:                Console.WriteLine("两个整数相除的结果：{0}/{1}={2}",number_1,
                       number_2,result);
18:            }
19:        }
20: }
```

以下是几次不同的运行结果。

（1）分别输入 100 和 7 的运行结果

```
请输入第一个整数：
100
请输入第二个整数：
7
两个整数相除的结果：100/7=14
```

（2）分别输入 100 和 0 的运行结果

```
请输入第一个整数：
100
请输入第二个整数：
0

未处理的异常：System.DivideByZeroException: 试图除以零。
在 CSHARP10_1.Program.Main(String[] args) 位置 G:\教材例题\CSHARP10-1\CSHARP10-1\Program.cs:行
号 16
```

本次运行中输入的除数为 0，这是一个无效的值。输出结果中显示未处理的异常，异常名为“System.DivideByZeroException”。原因是“试图除以零”，并在下一行显示产生异常的位置，即代码的第 16 行。产生异常也称为抛出一个异常。

（3）分别输入 100 和字符串“hello”的运行结果

```
请输入第一个整数：
100
请输入第二个整数：
hello
未处理的异常：System.FormatException: 输入字符串的格式不正确。
在 System.Number.StringToNumber(String str,NumberStyles options,NumberBuffe
r& number,NumberFormatInfo info,Boolean parseDecimal)
在 System.Number.ParseInt32(String s,NumberStyles style,NumberFormatInfo info)
在 CSHARP10_1.Program.Main(String[] args) 位置 G:\教材例题\CSHARP10-1\CSHARP10-1\Program.cs:行号
```

```
15
```

本次运行中输入的除数为一个字符串。这是一个非零的值，输出结果中显示未处理的异常，异常名为“System. FormatException”。原因是“输入字符串的格式不正确”，并在下一行显示产生异常的位置，即代码的第 14 行。

2. 使用 try 和 catch 捕捉和处理异常

要想捕获异常，需要将可能产生异常的语句放在 try 语句中。try 语句提供了在语句执行过程中捕获异常的机制。

在 C# 中，try 语句的使用格式有以下三种：

（1）try 后跟一个或多个 catch 块语句。

（2）try 后跟一个 finally 块语句。finally 块语句在后面介绍。

（3）try 后跟一个或多个 catch 块语句和一个 finally 块语句。

使用 try 和 catch 捕捉和处理异常的一般格式如下：

```
try
{
    语句块 1                        //可能引发异常的代码
}
catch(异常类型 1 异常对象 1)        //捕捉异常类 1 对象
{
    语句块 2                        //实现异常处理
}
catch(异常类型 2 异常对象 2)        //捕捉异常类 2 对象
{
    语句块 3                        //实现异常处理
}
…
```

try-catch 语句的执行逻辑如下：

当 try 块中有异常发生时，程序先创建一个包含异常信息的异常对象，然后从前到后依次搜索是否有与该异常对象匹配的 catch 代码块。找到匹配的代码块，就会执行该 catch 块中的语句，实现异常处理。如果未发生异常，则跳过 catch 子句，继续执行 try-catch 之后的语句。

例 10-2　修改例 10-1，添加捕捉和处理异常的功能。

本程序对可能出现的错误进行捕捉，然后对出现的错误进行处理，其中 try 语句使用后跟一个或多个 catch 块语句的格式，每个 catch 块语句捕获一个异常错误。程序代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

namespace CSHARP10_2
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("请输入第一个整数：");
                int number_1=Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("请输入第二个整数：");
                int number_2=Convert.ToInt32(Console.ReadLine());
                int result=number_1/number_2;
                Console.WriteLine("两个整数相除的结果：{0}/{1}={2}",
                        number_1,number_2,result);
            }
            catch(DivideByZeroException)
            {
                Console.WriteLine("除数不能为零");
            }
            catch (FormatException)
            {
                Console.WriteLine("第二个输入的不是数字");
            }
        }
    }
}
```

程序运行时，如果输入正确的数据，try 块中没有发生异常时，可以顺序完成 try 块中的所有语句，然后执行 try 与 catch 块后面的第一条语句，可以得出正确的结果。

分别输入 100 和 0 时，运行结果如下：

```
请输入第一个整数：
100
请输入第二个整数：
0
除数不能为零
```

分别输入 100 和字符串"hello"时，运行结果如下：

```
请输入第一个整数：
100
请输入第二个整数：
hello
第二个输入的不是数字
```

显然，两次错误的输入都被捕捉到了，并且对出现的错误进行了处理，这里的处理仅仅显示出错信息。

程序中的第 14～19 行是例 10-1 中的主要内容，是输入数据、计算和输出结果部分。本题中的第 12 行使用 try 语句将 14～19 行包围起来构成 try 语句块，块中的代码是可能出现异常以及发生异常时跳过的代码。如果块中的第 18 行计算出错，就不会执行第 19 行显示计算结果。

当 try 块中发生异常时，程序寻找第一个与异常匹配的 catch 块，相应的 catch 块会捕获并处理异常。本程序的 try 块后面有两个 catch 块，第 21～24 行为第一个 catch 块，catch 后面的括号用来指定异常参数，表示 catch 块可以处理的异常类型。如果不指定异常类型，则表示捕获所有的异常类型。

catch(DivideByZeroException)表示捕获除数为零的异常，catch 块内的语句：Console. WriteLine("除数不能为零")；表示对异常的处理，这里的处理是显示一个提示信息。

第 25～28 行为第二个 catch 块，捕获 FormatException 异常，对出现的异常也显示一个提示信息。

使用 catch 语句时，要注意下面的问题：

(1) 在 catch 语句中指定一个异常类型时，该类型必须是 System. Exception 或其派生类。

(2) catch 语句中可以不指定异常类型和异常对象名，这样的 catch 语句称为通用的 catch 语句。一个 try 块中只能有一条通用的 catch 语句，并且必须是该 try 块中最后一条 catch 语句，其格式为：catch{… }。

(3) try 和 catch 后面的一对花括号{}是必需的，即使代码块中只有一条语句也是如此。

(4) catch 语句中同时指定异常类型和异常对象时，该对象代表当前正在被处理的异常，可以在 catch 语句块内部使用。

(5) try 语句中可以有多个 catch 块，由于在搜索匹配异常时是按从前向后的顺序进行的，如果某个 catch 语句指定的类型与在此之前的某个 catch 指定的类型一致，或者是由此类型派生而来，则程序运行时会出现错误。

try 块和相应的 catch 与 finally 块构成 try 语句。这里要注意区别 try 块和 try 语句。try 块是指关键字 try 后面的代码块，try 语句包括从关键字 try 开始到最后一个 catch 与 finally 块的所有代码，包括 try 块和相关的 catch 与 finally 块。

10.2　.NET 的 Exception 层次

C# 名字空间 System 中的 Exception 类是.NET 框架的异常类机制中的所有异常基类，所有的异常必须用一个 System. Exception 类或其派生类的实例表示。例如，在例 10-2 中的 DivideByZeroException 和 FormatException 都是该类的派生类，该类有一个只读属性 Message，该属性用来描述出现异常的信息。

C＃中常见的系统异常类见表 10-1。

表 10-1　常见的系统异常类

异　常　类	引发异常的原因
AccessViolationException	试图读写受保护的内存
ApplicationException	发生非致命应用程序错误
ArithmeticException	算术运算、类型转换或转换操作错误
DivideByZeroException	试图用零除整数值或十进制数值
FieldAccessException	试图非法访问类中的私有字段或受保护字段
IndexOutOfRangeException	试图访问索引超出数组界限的数值
InvalidCastException	无效类型转换或显示转换
NotSupportedException	调用的方法不受支持
NullReferenceException	尝试取消引用空对象
OutOfMemoryException	没有足够的内存继续执行应用程序
OverflowException	在选中的上下文所执行的操作导致溢出
FileLoadException	找到托管程序集却不能加载它
FileNotFoundException	尝试访问磁盘上不存在的文件

例 10-3　修改例 10-2，使用基类异常捕获相关异常类型层次中的异常，使其可以显示异常的信息说明。

程序如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP10_3
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("请输入第一个整数：");
                int number_1=Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("请输入第二个整数：");
                int number_2=Convert.ToInt32(Console.ReadLine());
                int result=number_1/number_2;
```

```
19:                    Console.WriteLine("两个整数相除的结果：{0}/{1}={2}",
                           number_1,number_2,result);
20:                }
21:                catch (Exception E)
22:                {
23:                    Console.WriteLine(E.Message);
24:                }
25:            }
26:        }
27: }
```

程序中第 21～24 行定义 catch 块，其中的 E 是异常的对象，使用参数“Exception E”可以捕获所有从基类 Exception 派生的异常。异常处理中的 E. Message 是对象 E 的 Message 属性，表示异常信息的说明。第 23 行的异常处理只显示出错的内容。

分别输入 100 和 0 时，运行结果如下：

```
请输入第一个整数：
100
请输入第二个整数：
0
试图除以零。
```

分别输入 100 和字符串"hello"时，运行结果下：

```
请输入第一个整数：
100
请输入第二个整数：
hello
输入字符串的格式不正确。
```

两次结果中的“试图除以零”和“输入字符串的格式不正确”就是发生两种异常时 E. Message 属性的值。

如果对基类 Exception 的所有派生类类型要进行相同的处理，即使用相同的代码，可以在 catch 块中只写出基类 Exception，系统就可以捕获基类和所有的派生类，这样，代码比较简洁。

10.3 finally 语句块

在 try-catch 语句中，只有捕获到异常，才会执行 catch 语句中的代码。有一些特殊的操作，无论是否发生异常都必须执行。例如动态请求和释放资源，文件的关闭，内存的回收等，如果不执行这些操作，就会造成系统资源的占用和不必要的浪费。

对于这样一些无论是否捕获到异常都必须执行的代码，可以用 finally 关键字定义，将这些代码放在 finally 块中。

1. Finally 语句的使用

Finally 语句常常与 try-catch 语句配合使用，完整的使用格式如下：

```
try
{
    语句块 1                        //可能引发异常的代码
}
catch(异常类型 1 异常对象 1)         //捕捉异常类 1 对象
{
    语句块 2                        //实现异常处理
}
finally
{
    语句块 3                        //无论是否异常，都要进行处理
}
```

例 10-4 编写使用 finally 语句块的异常处理程序，程序的功能是打开一个文本文件，然后读出该文件的内容，最后将读出的内容显示出来，不论文件是否存在，都要执行文件的关闭操作。

程序代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;                          // 对文件操作时要使用 System.IO

namespace CSHARP10_4
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamReader file=null;       //定义文件流对象 file
            try
            {
            file=new StreamReader(new FileStream(@"g:\123.txt",
                FileMode.Open));          //打开文件
                Console.WriteLine(file.ReadLine());          //读出文件内容并显示
            }
            catch (FileNotFoundException)   //捕获文件不存在的异常
            {
                Console.WriteLine("文件没有找到");
```

```
22:                 }
23:                 finally
24:                 {
25:                     if (file != null) file.Close();
26:                     Console.WriteLine("finally中的语句被执行,文件被关闭");
27:                 }
28:             }
29:         }
30: }
```

本例只涉及文件的简单操作。代码的第 13 行定义文件流对象 file,第 16 行用来打开文本文件"g:\123.txt",第 17 行用来从文件中读出内容,第 25 行用来关闭文件。这几条语句的详细内容见文件一章,本章只要知道其功能即可。

假定在 G:盘中有一个文本文件 123.txt,其内容为一行字符“hello”,见图 10-1。



图 10-1　文本文件的内容

程序中第 14～18 行为 try 块,其中有两条语句。第 16 行为打开文件“123.txt”,第 17 行从该文件中读出一行字符串并输出显示。如果该文件存在,则程序的执行结果如下:

```
hello
finally中的语句被执行,文件被关闭
```

如果该文件不存在,则程序显示如下:

```
文件没有找到
finally中的语句被执行,文件被关闭
```

第 19～22 行的 catch 块捕获文件不存在的异常“FileNotFoundException”。如果捕获到,则显示"文件没有找到"的信息。

第 23～27 行是 finally 块,其中含有两条语句,分别是关闭文件和显示信息,不论异常是否捕获到,这两条语句都要执行。

2. finally 块的返回

finally 块结束之后执行的下一条语句取决于异常处理的状态。如果 try 顺利完成,或 catch 块捕获并处理了异常,则程序转入执行 finally 块后面的下一条语句。如果异常没有捕获或 catch 块再次抛出异常,则程序控制转入外层 try 块。外层的 try 块可能在调用方法中,也可能在其调用者中。

try 块中可以嵌套 try 块,这里外层 try 语句的 catch 块处理内层 try 中没有捕获的任何异常。如果 try 块执行并有相应的 finally 块,则执行 finally 块,即使 try 块因为 return 语句而终止。

使用 finally 语句要注意下面的问题:

(1) finally 语句中不能出现 return 语句,return 语句在执行 finally 块之后执行。

(2) 可以省略 catch 块,即使用 try-finally 结构,这时不对异常进行处理。

10.4 using 语句

资源释放代码要放在 finally 块中,不管相应的 try 块中使用资源时是否发生异常,都要保证释放资源。在 try 块中使用资源和在相应的 finally 块中释放资源的代码可以使用 using 语句进行简化。

例如,对于下面的 try 和 catch 块的组合:

```
{
    exampleObject.SomeMethod();
    finally
    {
        If(exampleObject!=null)
            ((Idisposable)exampleObject).Dispose();
    }
}
```

上面的代码中,资源可以使用下面的 using 语句进行简化:

```
using (ExampleObject exampleObject=new ExampleObject())
{
    exampleObject.SomeMethod();
}
```

using 语句隐式将代码放在 try 块中,相应的 finally 块中调用其 Dispose 方法。这里的 using 与使用名字空间的 using 指令没有关系。

10.5 throw 语句与抛出异常

前面各个例题中捕获到的异常,都是当程序遇到错误时,由系统自动通知运行环境异常的发生,有时也可以在代码中手动告诉运行环境在什么时候发生了什么异常。手动抛出异常可以使用 throw 语句,其格式如下:

```
throw [异常对象]
```

省略异常对象时,该语句只能用在 catch 语句中,用于再次引发异常处理。

语句中带有异常对象时,则抛出指定的异常类,并显示异常的相关信息。该异常既可以是预定义的异常类,也可以是自定义的异常类。如果是自定义的异常类,则应该从 ApplicationException 类派生。

通过 throw 语句可无条件地主动在程序中抛出异常,抛出的异常要用 catch 语句捕获。

例 10-5 使用 throw 语句抛出异常,用来处理使用数组时下标出界的错误。程序运

行时，输入一个下标值，输出该下标对应的元素值。当输入的下标不在正确范围时，抛出异常，显示提示信息。

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5:
 6: namespace CSHARP10_5
 7: {
 8:     class Program
 9:     {
10:         static void Main(string[] args)
11:         {
12:             int [] array= {0,1,2,3,4,5,6,7,8,9};
13:             int index;
14:             try
15:             {
16:                 Console.WriteLine("请输入下标");
17:                 index= Convert.ToInt32(Console.ReadLine());
18:                 if (index< 0 || index > array.Length - 1)
19:                     throw new IndexOutOfRangeException("下标应在 0到"+
                             Convert.ToString(array.Length - 1)+ "之间");
20:                 Console.WriteLine("数组中第{0}个元素为：{1}",index,
                           array[index]);
21:             }
22:             catch (IndexOutOfRangeException e)
23:             {
24:                 Console.WriteLine(e.Message);
25:             }
26:         }
27:     }
28: }
```

输入下标正确时程序的运行结果如下：

```
请输入下标
4
数组中第 4个元素为：4
```

输入的下标超界时程序的运行结果如下：

```
请输入下标
10
下标应在 0到 9之间
```

第 18～19 行当输入的下标值不在正常范围内时抛出异常错误。错误类型为下标超

出范围异常，即 IndexOutOfRangeException。

第 22～25 行捕获该异常。如果捕获到，则显示相应的信息。

10.6 Exception 类的常用属性

Exception 是异常的基类型，该类有以下几个常用的属性，用于形成错误的消息，表示捕获的异常。

1. Message

Message 属性存储与 Exception 对象相关的错误消息。这个消息可以是与异常类型相关联的默认消息，或抛出 Exception 对象时传入 Exception 对象构造函数的定制消息。

在例 10-5 中，第 24 行语句 Console.WriteLine(e.Message)；使用该属性显示出错的信息。

2. StackTrack

该属性包含的字符串表示方法调用堆栈。StackTrack 表示系列方法在发生异常时还没有处理完毕。

堆栈踪迹显示发生异常时的完整方法调用堆栈，堆栈中的信息包括发生异常时调用堆栈中的方法名、定义这些方法的类名和定义这些类的名字空间。

3. innerException

InnerException 属性返回与传递给构造函数的值相同的值。如果没有向构造函数提供内部异常值，则返回 null 引用。此属性为只读。

4. 其他 Exception 属性

其他 Exception 属性包括 HelpLink、Source 和 TargetSite。HelpLink 属性指定描述所发生问题的帮助文件的地址。如果没有这个文件，则该属性为 null。

Source 属性指定发生异常的程序名。TargetSite 属性指定产生异常的方法。

10.7 用户定义异常类

大多数情况下，可以使用.NET 框架类库中现有的异常类处理程序中发生的异常。有时需要针对程序中发生的问题创建新的异常类，这就是用户定义的异常类。用户定义异常类要直接或间接继承 System 名字空间中的 Exception 类，类名以 Exception 结尾。

例 10-6 创建用户自定义的 NegativeNumberException 类，用来处理对负数进行平方根运算时产生的异常。

程序代码如下：

```
1: using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CSHARP10_6
{
    class NegativeNumberException:Exception
    {
        public NegativeNumberException():base("Illegal operation for a negative number")
        {    //类体为空
        }
        public NegativeNumberException(string messageValue):base(messageValue)
        {    //类体为空
        }
        public NegativeNumberException(string messageValue,Exception inner):base(messageValue,inner)
        {    //类体为空
        }
    }
    class Program
    {
        public static double SquareRoot(double value)
        {
            if(value< 0)
                throw new NegativeNumberException ("Square root of negative number not permitted");
            else
                return Math.Sqrt(value);
        }
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("请输入一个实数");
                double x= Convert.ToDouble(Console.ReadLine());
                double result= SquareRoot(x);
                Console.WriteLine("{0}的平方根为：{1}",x,result);
            }
            catch (FormatException formatExceptionParameter)
            {
                Console.WriteLine(formatExceptionParameter.Message);
            }
```

```
42:                 catch (NegativeNumberException negativeNumberException-
                         Parameter)
43:                 {
44:                     Console.WriteLine(negativeNumberExceptionParameter.Message);
45:                 }
46:             }
47:         }
48: }
```

程序运行时，如果输入大于等于零的数，则输出如下正确的结果：

```
请输入一个实数
5
5的平方根为：2.23606797749979
```

如果输入一个负数，例如输入－5，则输出结果如下：

```
请输入一个实数
-5
Square root of negative number not permitted
```

如果输入一个非数值，例如输入一个字符串，则输出结果如下：

```
请输入一个实数
hello
输入字符串的格式不正确。
```

第 8～19 行是用户自定义的异常类 NegativeNumberException，处理程序对负数进行非法运算(例如计算平方根)时产生的异常。该类中定义了 3 个构造函数，一个无参构造函数(第 10～12 行)；一个用来接收错误消息字符串参数的构造函数(第 13～15 行)，还有一个用来接收错误消息字符串和 Exception 变量两个参数的构造函数(第 16～18 行)，该类的基类是 Exception。

第 22～28 行定义 SquareRoot 方法，用来计算参数的平方根。如果参数是大于等于零的数值，则调用 Math 类中的 Sqrt 方法计算平方根。如果参数是负数，则方法中抛出 NegativeNumberException 异常。

第 31～37 行的 try 块中输入实数，然后调用 SquareRoot 方法计算参数的平方根。如果输入的不是有效数值，则发生 FormatException 异常，第 38～41 行处理这个异常。

如果输入负数，则 SquareRoot 方法抛出 NegativeNumberException 异常，第 42～45 行处理这个异常。

习题

1. 用继承创建异常基类和各种异常派生类，编写程序演示指定基类的 catch 可以捕获派生类异常。

2. 编程演示异常处理器顺序的重要性。演示一个正确顺序的 catch 块,即基类异常处理器放在所有派生类异常处理器之后,再演示一个错误的 catch 块,即基类异常处理器放在所有派生类异常处理器之前。这时编译第二个程序会发生什么情况?

3. 编程演示再次抛出异常。

4. 用异常表示构造对象时发生的问题,编程显示构造函数将构造函数故障信息传递给异常处理器。抛出的异常也应包含发给构造函数的形参。

5. 编程输入里程数和耗油量,计算每升的里程数。程序中使用异常处理器。当输入的里程数或耗油量无法转换成 double 值时,处理 FormatException。

第11章 图形界面编程

11.1 Windows 编程概述

Windows 编程遵循事件驱动的程序设计思想。在事件驱动的程序结构中，程序的控制流程不再由事件的预定发生顺序来决定，而是由实际运行时各种事件的实际发生来触发，而事件的发生可能是随机的、不确定的，并没有预定的顺序。事件驱动的程序允许用户用各种合理的顺序来安排程序的流程。对于需要用户交互的应用程序来说，事件驱动的程序设计有着传统程序设计方法无法替代的优点。事件驱动是一种面向用户的程序设计方法，在程序设计过程中除了完成所需要的程序功能之外，更多地考虑了用户可能的各种输入（消息），并有针对性地设计相应的处理程序。事件驱动程序设计也是一种“被动”式的程序设计方法。程序开始运行时，处于等待消息状态，然后取得消息并对其作出相应反应，处理完毕后又返回处于等待消息的状态。事件驱动原理的程序的工作流程如图 11-1 所示。

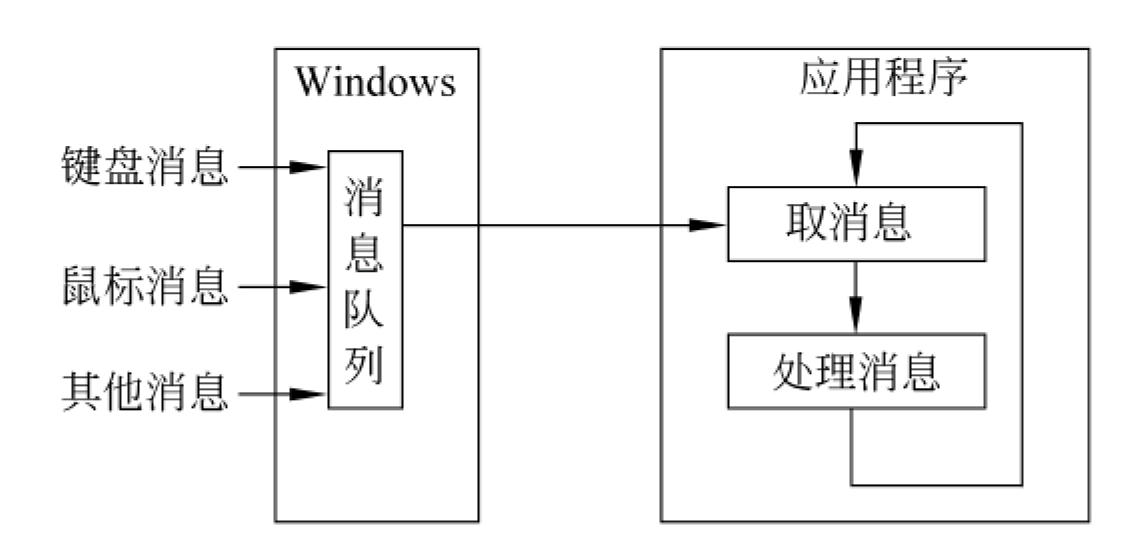


图 11-1 事件驱动原理的程序的工作流程

事件驱动围绕着消息的产生与处理展开，事件驱动是靠消息循环机制来实现的。消息是一种报告有关事件发生的通知。Windows 应用程序的消息来源有以下四种。

（1）输入消息：包括键盘和鼠标的输入。这一类消息首先放在系统消息队列中，然后由 Windows 将它们送入应用程序消息队列中，由应用程序来处理消息。

（2）控制消息：用来与 Windows 的控制对象，如列表框、按钮、检查框等进行双向通信。当用户在列表框中改动当前选择或改变了检查框的状态时发出此类消息。这类消息一般不经过应用程序消息队列，而是直接发送到控制对象上去。

（3）系统消息：对程序化的事件或系统时钟中断作出反应。一些系统消息，如 DDE

消息(动态数据交换消息)要通过 Windows 的系统消息队列;而有的则不通过系统消息队列而直接送入应用程序的消息队列,如创建窗口消息。

(4) 用户消息:是程序员自己定义并在应用程序中主动发出的,一般由应用程序的某一部分内部处理。

11.1.1　窗体

窗体是应用程序的编程窗口和对话框。窗体可包含多个控件,就像包含多个控件的容器。几乎每个应用程序都要包含一个窗体。如果一个应用程序中需要包含多个窗体,就必须给它们取不同的名称。对于用户来说,窗体的界面就是应用程序,程序的可用性完全依赖于窗体界面。掌握主窗体常用项目的设置是非常重要的,有利于快速按照要求构建应用程序框架。

11.1.2　事件处理

Windows 利用事件(或称之为消息)驱动程序运行。事实上,多数程序都是事件驱动的——即执行流程是由外界发生的事件所确定的。事件是一个信号,它告知应用程序有重要情况发生。例如,用户单击窗体上的某个控件时,窗体引发一个 Click 事件并调用一个处理该事件的过程。事件还允许在不同任务之间进行通信。比方说,应用程序脱离主程序执行一个排序任务。若用户取消这一排序,应用程序可以发送一个取消事件让排序过程停止。

理解 Windows 的事件驱动是理解 Windows 程序运行机制的关键。Windows 编程使用事件驱动的程序设计思想。在事件驱动的程序结构中,程序的控制流程不再由事件的预定发生顺序来决定,而是由实际运行时各种事件的实际发生来触发,而事件的发生可能是随机的、不确定的,并没有预定的顺序。事件驱动的程序允许用户用各种合理的顺序来安排程序的流程。事件驱动是一种面向用户的程序设计方法,在程序设计过程中除了完成所需要的程序功能之外,更多地考虑了用户可能的各种输入(消息),并有针对性地设计相应的处理程序。事件驱动程序设计也是一种“被动”式的程序设计方法,程序开始运行时,处于等待消息状态,然后取得消息并对其作出相应反应,处理完毕后又返回等待消息的状态。下面用一个例子来进一步对事件进行说明。

例 11-1　创建含有两个文本框的例子。在第一个文本框中键入字符时,第二个文本框的内容始终和第一个文本框的内容保持一致。

当在一个文本框中键入字符时,会触发事件——TextChanged。为该事件编写代码,将第一个文本框中的内容复制到第二个文本框中。这样就达到了目的。按照以下步骤来完成程序:

(1) 开始一个新的 Windows 项目,命名为 CSHARP11-1。在 Form1 窗体上放置两个文本框控件,同时放置两个 Label 标签标识这两个文本框,并设置属性如表 11-1～表 11-4 所示。将控件放置到窗体上的办法是:从控件工具箱中选择需要的控件,拖动到窗体上即可。选中控件后可以调整控件的大小。调整好控件的大小和位置后,窗体看起来如图 11-2 所示。

表 11-1　Label1 控件的属性

属性	值
Name	LblInput
Text	输入的字符

表 11-2　Label2 控件的属性

属性	值
Name	LblCopy
Text	复制的字符

表 11-3　TextBox1 控件的属性

属性	值
Name	textInput

表 11-4　TextBox2 控件的属性

属性	值
Name	textCopy
ReadOnly	True

(2) 在输入字符的 textBox 上单击右键,重新打开“属性”对话框,然后切换到消息列表。如图 11-3 所示,可以看到在这里边列出了所有 Windows 已经预先定义好的、textInput 控件可以处理的事件。



图 11-2　StringCopy 程序的界面

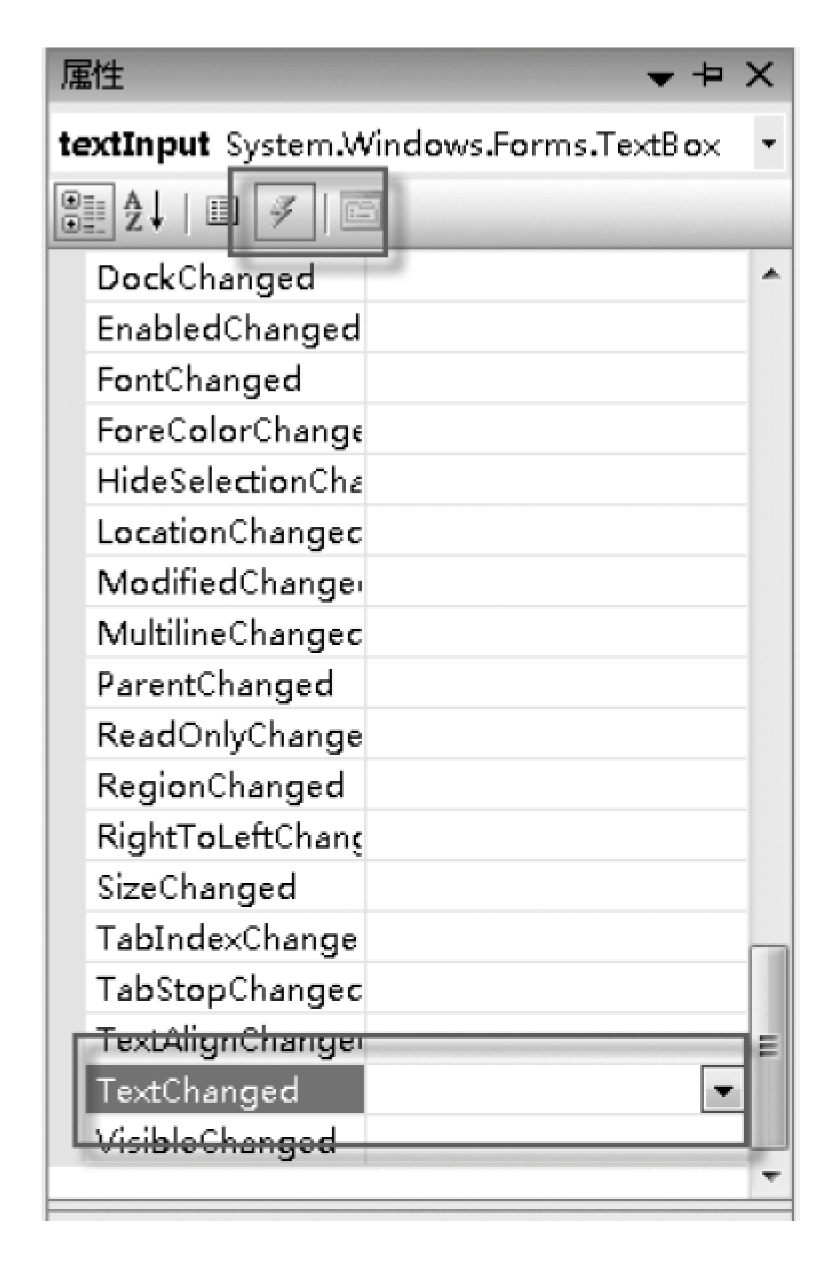


图 11-3　事件与属性的选择

拉动右边的滑块,选择 TextChanged,这是将要处理的事件。在 TextChanged 的右侧双击添加消息处理程序,Visual Studio 2013 自动在代码窗口添加了处理该事件的代码框架。在事件处理方法中添加代码:

```
private void textInput_TextChanged(object sender,EventArgs e)
{
    textCopy.Text=textInput.Text;
}
```

由于所举的例子较为简单,所以整个程序只需要添加这一行代码就够了。现在运行

程序,可以看到,在文本框 Input 中发生任何改变,文本框 Copy 始终和它保持一致。

上面的程序是如何工作的呢?实际情况是这样的:每当在文本框 Input 中输入字符或者删除字符时,该文本框的内容发生改变,此时,系统(Windows)发送 TextChanged 消息到应用程序。由于存在该消息的处理程序,应用程序将调用该处理程序,因此语句:

```
textCopy.Text=textInput.Text;
```

被执行,文本框 Input 中的内容被复制到文本框 Copy 中。

11.2 常用控件 1

控件是对数据和方法的封装。控件可以有自己的属性和方法。属性是控件数据的简单访问者。方法则是控件的一些简单而可见的功能。

使用现成的控件来开发应用程序时,控件工作在两种模式下:设计时态和运行时态。在设计时态下,控件显示在开发环境下的一个窗体中。设计时态下控件的方法不能被调用,控件不能与最终用户直接进行交互操作,也不需要实现控件的全部功能。运行状态下,控件工作在一个确实已经运行的应用程序中。控件必须正确地将自身表示出来,它需要对方法的调用进行处理并实现与其他控件之间的有效协同工作。

11.2.1 控件的属性和布局

控件的属性控制着对象的外观和行为。通过对同样的控件设置不同的属性,可以使它们表现出不同的外观和行为。许多属性是每一个控件都有的,还有一些属性是大部分控件都有的。这些属性对每一个控件来讲,用法是相同的。

将控件添加到窗体后,通常要设置控件的一个或多个属性。对于例 11-1 所添加的文本框控件,要设置它的 Name 和 Text 属性。

Name 属性非常重要。在程序代码中,它用来指明控件。由于程序一般有多个同类控件,所以可以用控件的 Name 属性来唯一标识某一特定的控件。每个控件都必须有名称,其名称用控件的 Name 属性值来表示。除此之外,特定窗体上的每个控件都必须有一个唯一的名称。Visual C# 为窗体上放置的每一个控件都分配了一个默认名称。如 TextBox1、TextBox2 等。更改默认控件的名字使其更加具有现实意义,是一个良好的编程习惯。

在属性窗口中可以设置控件的属性。被选中的控件名字出现在属性窗口上部的下拉框中,该控件的属性被分类后列出。左边一栏是属性的名字,右边是属性的值。单击属性的名字,可以在下方看到对该属性的简单提示。要改变属性的值,只需单击原有属性的值,作相应的改变即可。

控件在窗体上应当排列整齐。要移动控件,只需选中该控件,用鼠标拖动即可。如果想更改控件尺寸,首先必须选中它(用鼠标单击一个控件即可选中),令其可缩放的控制点显示出来。然后利用鼠标,通过拖动控制点的方式,更改控件的尺寸。控件顶部和底部边缘的控制点用于更改控件的高度,左边和右边的控制点用于更改控件的宽度,控件四角的

控制点同时更改其高度和宽度。

默认状态下，有的控件(如文本框控件)只有两个控制点是可用的(一共有 8 个)，其余的呈灰色不可用。这是因为文本框控件的 AutoSize 属性在默认状态下被设置为 True。AutoSize 属性会根据控件即将显示的文本字体大小自动调整文本框的高度。因此，调整高度大小的控制点不可用。如果需要控制文本框控件的高度，可以将 AutoSize 属性设置为 False，此时 8 个控制点就全部可用了。

另外，还可以在属性窗口中分别修改 Size 和 Location 属性，从而修改对象的大小和位置。Size 属性有两个值分别表示控件的高度和宽度。Location 的两个值分别表示控件相对于容器的 x、y 坐标。



图 11-4 “对齐”控件的菜单

如果控件不是对得很齐，还可以这样做：将要对齐的控件选中。为了做到这一点，可以先选中一个，再按住 Ctrl 键用鼠标选中其余的。例如，要对齐 lblInput 标签控件和位于它旁边的 txtInput 文本框控件。首先用上述方法将这两个控件选中，然后在菜单中选择“格式”→“对齐”→“中间”对齐，如图 11-4 所示。

11.2.2 卷标、文本框和按钮

控件是可以和用户或程序实现互动的一个对象。绝大多数程序都是可以互动的——它们需要从用户那里获取信息，并向用户反馈信息。基于 Windows 的程序通过程序窗体的控件实现与用户的交互。正如例 11-1 通过文本框控件得到用户输入的字符，同时利用另一个文本框控件将结果显示给用户。Visual C# 的许多控件都可以用于获取用户信息。在前面的例子中，用到了以下最常用的 3 种控件。

1. Label(标签)控件

为控件和窗体的其他组成部分提供标识。使用 Label，可以给用户提供出窗体功能的有关信息。从广义上说，窗体中的每一条文字都是一个 Label 控件。Label 控件通常用于提供控件的描述性文字。例如，可以使用 Label 向 TextBox 控件添加描述性文本，以通知用户有关控件期望的数据类型。Label 控件还可用于向窗体添加描述性文本，以提供有用的信息。可以将 Label 添加到 Form 的顶部，后者为用户提供如何在窗体上的控件中输入数据的说明。Label 控件还可用来显示应用程序状态的运行时信息。例如，可将 Label 控件添加到窗体，以便在处理一列文件时显示每个文件的状态。Label 控件由 Label 类描述，Label 类继承自 Control 类，在名字空间：

```
System.Windows.Forms
```

中。Label 类中经常用到的属性如下：

- AutoEllipsis：获取或设置一个值，指示是否要在 Label 的右边缘显示省略号(…)以表示 Label 文本超出 Label 的指定长度。

- AutoSize：获取或设置一个值，该值指示是否自动调整控件的大小以显示其完整内容。
- BorderStyle：获取或设置控件的边框样式。
- Image：获取或设置显示在 Label 上的图像。
- Text：获取或设置与此控件关联的文本。
- TextAlign：获取或设置标签中文本的对齐方式。

2. TextBox（文本框）控件

一个应用程序中会多次用到该控件。TextBox 控件的应用范围非常广，例如可用来显示一个由多行文本组成的版本信息。实际上 TextBox 能容纳的文本数量是没有限制的，当文本数量超出文本框的尺寸时，文本框还会添加自己的滚动条。TextBox 和 Label 控件之间的差别在于：TextBox 控件中的文本可以被编辑，而 Label 控件中的文本不能被编辑。

利用 TextBox 控件，用户可以在应用程序中输入文本，包括多行编辑和密码字符屏蔽。通常，TextBox 控件用于显示单行文本或将单行文本作为输入来接受。可以使用 Multiline 和 ScrollBars 属性，从而能够显示或输入多行文本。通过将 AcceptsTab 和 AcceptsReturn 属性设置为 true，可在多行 TextBox 控件中更加灵活地操作文本。TextBox 类继承自 TextBoxBase 类，后者同 label 一样，派生自 Control 类。TextBox 类所在的名字空间和 Label 类是一样的：System. Windows. Forms。TextBox 类经常用到的属性有：

- AcceptsReturn：获取或设置一个值，该值指示在多行 TextBox 控件中按 Enter 键时，是在控件中创建一行新文本还是激活窗体的默认按钮。
- AcceptsTab：获取或设置一个值，该值指示在多行文本框控件中按 Tab 键时，是否在控件中键入一个 Tab 字符，而不是按选项卡的顺序将焦点移动到下一个控件。
- BackColor：获取或设置控件的背景色。
- Focused：获取一个值，该值指示控件是否有输入焦点。
- Font：获取或设置控件显示的文字的字体。
- Multiline：获取或设置一个值，该值指示此控件是否为多行 TextBox 控件。
- PasswordChar：获取或设置字符，该字符用于屏蔽单行 TextBox 控件中的密码字符。
- ReadOnly：获取或设置一个值，该值指示文本框中的文本是否为只读。
- RightToLeft：获取或设置一个值，该值指示是否将控件的元素对齐以支持使用从右向左的字体的区域设置。
- SelectedText：获取或设置一个值，该值指示控件中当前选定的文本。
- Text：获取或设置 TextBox 中的当前文本。
- TextAlign：获取或设置 TextBox 控件中文本的对齐方式。
- TextLength：获取控件中文本的长度。

- WordWrap：指示多行文本框控件在必要时是否自动换行到下一行的开始处。

在 TextBox 中经常用到的事件有：

- Click：在单击文本框时发生。
- Enter：进入控件时发生。
- GotFocus：在控件接收焦点时发生。
- KeyDown：在控件有焦点的情况下按下键时发生。
- KeyUp：在控件有焦点的情况下释放键时发生。
- Leave：在输入焦点离开控件时发生。
- LostFocus：在控件失去焦点时发生。
- TextChanged：在 Text 属性值更改时发生。

3. Button(按钮)控件

用户可以单击按钮控件触发程序动作。Button 类从 ButtonBase 类继承，后者继承自 ContentControl 类。而 ContentControl 类和其他大多数控件一样，是从 Control 类派生的。Button 类常用的属性有：

- ClickMode：获取或设置 Click 事件何时发生。
- IsCancel：获取或设置一个值，该值指示 Button 是否是一个“取消”按钮。用户可以通过按 Esc 键激活“取消”按钮。
- IsDefault：获取或设置一个值，该值指示 Button 是否是一个默认按钮。用户可以通过按 Enter 键调用默认按钮。

其中，ClickMode 有 3 种模式：(1)悬停，当鼠标悬停在该按钮上时激发此按钮；(2)按下，当鼠标指针位于该按钮上按下鼠标时激发此按钮；(3)释放，在该按钮上按下和释放鼠标后，才会激发此按钮。

Button 最主要的一个事件就是 Click 事件了，配合 ClickMode，在鼠标单击按钮时发生该事件。

11.2.3 组框、面板、复选框和单选钮

几乎每个程序都需要为用户提供选项，选项的作用是根据用户的实际情况，在程序运行时调整程序的状态或行为。下面介绍常用的 GroupBox、Panel、CheckBox 和 Radio Button 控件的使用。

RadioButton(单选按钮)控件用来让用户在一组选项中选定一项且只能选定一项。窗体内仅有一组选项按钮控件时，将它们直接放置在这个窗体内即可。当有两组或多组选项时，RadioButton 应该被放置到一个 GroupBox(组框)控件(下面将要介绍)内。

在窗体或框架内创建一个 RadioButton 时，在 Visual Studio 的工具箱内单击 RadioButton 对象，鼠标箭头变为十字形状。将鼠标移至窗体上或框架内的合适位置，按住鼠标左键，拖动鼠标。到适当大小时，释放鼠标左键即可。

RadioButton 有许多属性，其中最常用的如下：

- Text 属性：设定 RadioButton 旁边的文本内容。

- CheckAlign 属性：设定控件按钮与文本的位置关系。
- Checked 属性：由 Checked 属性设定 RadioButton 的状态。
  - True：RadioButton 被选定。
  - False：RadioButton 未被选定，默认设置。

RadioButton 最常见的事件是 CheckedChanged，每次选中或者不选中单选按钮时产生。在 Visual Studio 的设计模式下，双击单选按钮控件时，会自动添加该事件的事件处理程序。

CheckBox（复选框）控件是让用户在一组选项中选定一项或选定多项。窗体内仅有一组 CheckBox 控件时，可将它们放置在这个简单的窗体内；当有两组或多组 CheckBox 时，CheckBox 通常也被放置到一个 GroupBox 控件内。

CheckBox 的属性中也有 Text 属性、CheckAlign 属性和 Name 属性等，这些属性的用法几乎每个控件都大同小异。CkeckBox 最重要的属性是 Checked 属性。通过 Checked 属性可以检查或设定 CheckBox 是否被选中。

- Checked＝True，被选中。
- Checked＝False，未被选中。

CheckBox 中还有一个 CheckState 属性用来指示 CheckBox 目前的状态：

- CheckState＝Checked，被选中状态。
- CheckState＝UnChecked，未被选中状态。
- CheckState＝Indeterminate，不可用状态（当 ThreeState 属性设置为 true 时，有效）。

复选框的常见事件有：

- CheckedChanged：当复选框的 Checked 属性发生变化时产生。这是复选框的默认事件，也就是在设计视图双击复选框控件时，会产生该事件的空白方法。
- CheckStateChanged：当复选框的 CheckState 属性发生变化时产生。

RadioButton 控件在一组中只能选定一项，怎样对 RadioButton 控件分组呢？其中一个方法就是利用 GroupBox 控件（GroupBox 控件在工具箱的“容器”组中）。可以先将一个 GroupBox 控件放置在窗体上，然后将 RadioButton 控件放在 GroupBox 控件即可。在一个 GroupBox 控件中的 RadioButton 控件自动成为一组。另外，还可以设定 GroupBox 的 Text 属性，对分组进行说明。

CheckBox 也可以放在一个 GroupBox 控件中，但 CheckBox 并没有 RadioButton 分组的类似概念。将 CheckBox 放入一个 GroupBox 的作用主要是说明和装饰。

和 GroupBox 类似，用于在程序界面中安排控件的容器类控件还有 Panel（面板）。也可以将一组控件放入到 Panel 中。设计时，当移动组框或者面板时，其中的所有控件也会跟着一起移动；而且组框和面板还可以用来同时显示和隐藏一组控件，只需要改变组框和面板的 Visible（可见）属性就可以了。

组框和面板的主要不同在于：组框可以显示标题（文本）但是没有滚动条，而面板可以包含滚动条却没有标题。在默认情况下，组框的边框较细，面板可以通过 BorderStyle 属性改变边框。

组框常用的属性有：

- Controls：组框包含的控件集。
- Text：在组框的顶部显示的标题文本。

面板常用的属性有：

- AutoScroll：当面板由于太小而无法显示所有的控件时，这个属性指定是否出现滚动条。默认值为 false。
- BorderStyle：设置面板的边界样式。默认值为 None。其他选项有 Fixed3D 和 FixedSingle。
- Control：和组框类似，表示面板包含的控件集。

例 11-2 在这个例子中，利用上述控件组织用户界面，并从用户处收集一些数据。设计的界面如图 11-5 所示。



图 11-5 控件的使用与演示

在图 11-5 中，1 的地方使用了一个 Label，对下面一组控件进行了说明；2 是一个组框，框定了 4 个单选按钮（图中的 3）。这 4 个单选按钮形成了一组，在这 4 个单选按钮中只能选择一个。同样，后面还有“价格”和“运营商”两组单选按钮。

这 3 个组框被放在了一个面板（图 11-5 中的 4）中。特别注意的是“价格”这组单选按钮的长度超出了面板的高度，所以某些价格被遮挡了无法看见。这时，可以设定面板的属性 AutoScroll 为 True，当有控件超出时，会自动添加滚动条（如图 11-6）。图 11-5 中的 5 仍然是一个面板，读者可以试试，拖动面板，可以带动面板上的所有控件一起移动，这样便于更好地组织界面。图 11-5 中的 6 是一组复选框，允许用户有多个选择。

此时运行程序，如图 11-7 所示。拖动滚动条，可以看到“价格”的其余部分。

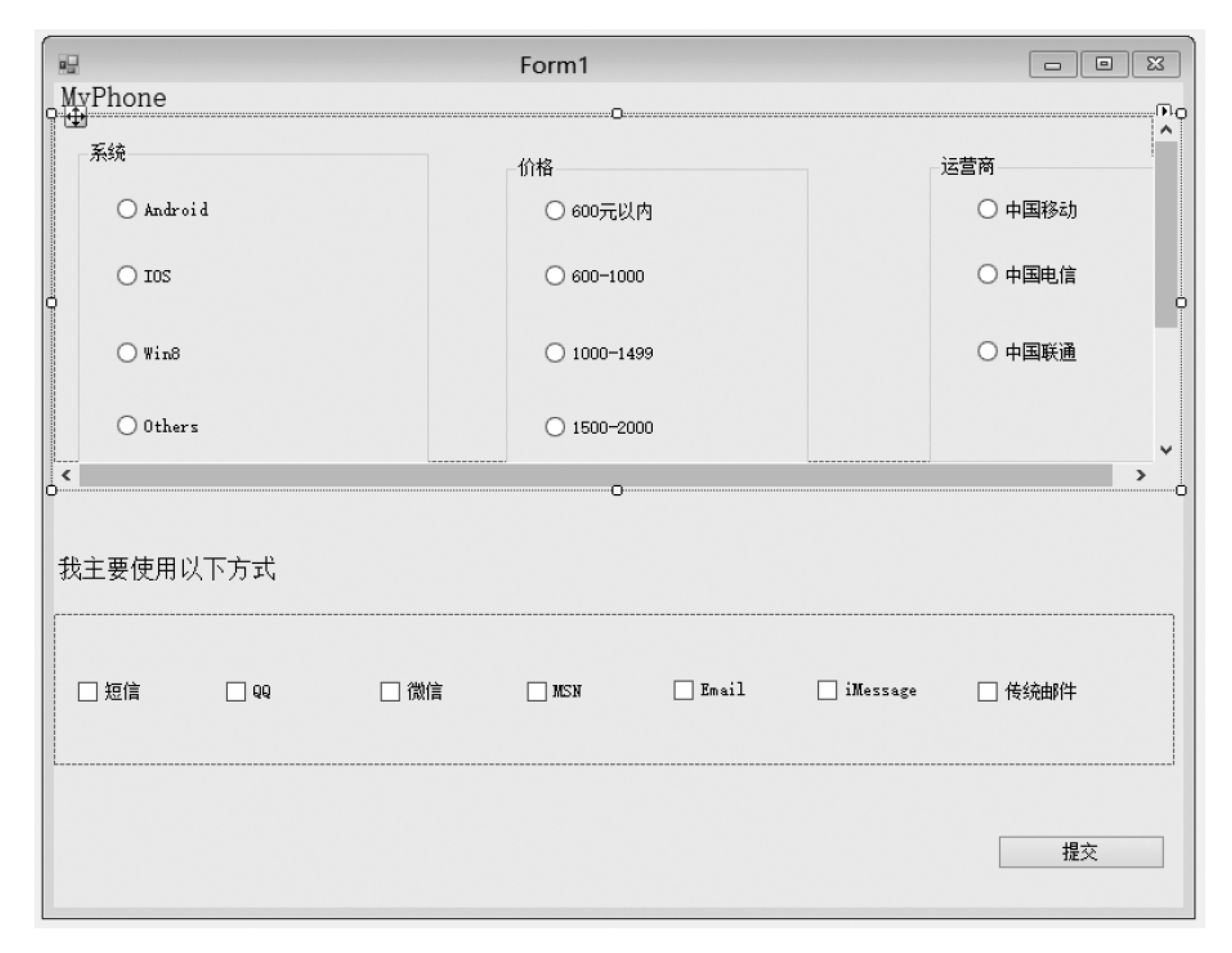


图 11-6　面板的 AutoScroll 为 True



图 11-7　设置滚动条

下面，为这个程序添加一些事件代码作为测试。首先，添加按钮的 Click 事件代码。按下按钮后，收集用户在单选按钮中的选择(为避免代码过多重复，没有收集复选框的选择)，并整理为一条信息通过一个 MessageBox 展现出来。在添加代码前，将各个控件的 name 属性修改得更有意义一些，而不是缺省的 radio1Button1、radioButton2 之类。整个程序的代码如下，其中 Button 的 Click 事件处理在 13～48 行。

```
01: using System;
02: using System.Windows.Forms;
03:
04: namespace CSHARP11_2
05: {
06:     public partial class Form1 : Form
07:     {
08:         public Form1()
09:         {
10:             InitializeComponent();
11:         }
12:
13:         private void submitButton_Click(object sender,EventArgs e)
14:         {
15:             string messages= "MyPhone: 系统 ";
16:             if (androidRadioButton.Checked)
17:                 messages+ = "Android;";
18:             if (iosRadioButton.Checked)
19:                 messages+ = "IOS;";
20:             if (win8RadioButton.Checked)
21:                 messages+ = "Win8;";
22:             if (othersRadioButton.Checked)
23:                 messages+ = "Others;";
24:
25:             messages+ = " 价格 ";
26:             if (low600RadioButton.Checked)
27:                 messages+ = "< 600;";
28:             if (btw600_1000RadioButton6.Checked)
29:                 messages+ = "600- 1000;";
30:             if (btw1001_1500RadioButton.Checked)
31:                 messages+ = "1001- 1500;";
32:             if (btw1501_2000RadioButton.Checked)
33:                 messages+ = "1501- 2000;";
34:             if (btw2001_2500RadioButton.Checked)
35:                 messages+ = "2001- 2500;";
36:             if (btw2501_3000RadioButton.Checked)
37:                 messages+ = "2501- 3000;";
38:
```

```
39:                 messages+=" 运营商 ";
40:                 if (chinaMobileRdioButton.Checked)
41:                     messages+="中国移动";
42:                 if (chinaTelecomRadioButton.Checked)
43:                     messages+="中国电信";
44:                 if (chianUnicomRadioButton.Checked)
45:                     messages+="中国联通";
46:
47:                 MessageBox.Show(messages);
48:             }
49:
50:             int[] check=new int[7];
51:             private void CheckBox_CheckedChanged(object sender,EventArgs e)
52:             {
53:                 if (sender==shortMessageCheckBox)
54:                     check[0]=shortMessageCheckBox.Checked ? 1 : 0;
55:                 if (sender==qqCheckBox)
56:                     check[1]=qqCheckBox.Checked ? 1 : 0;
57:                 if (sender==weChatCheckBox)
58:                     check[2]=weChatCheckBox.Checked ? 1 : 0;
59:                 if (sender==msnCheckBox)
60:                     check[3]=msnCheckBox.Checked ? 1 : 0;
61:                 if (sender==emailCheckBox)
62:                     check[4]=emailCheckBox.Checked ? 1 : 0;
63:                 if (sender==iMessageCheckBox)
64:                     check[5]=iMessageCheckBox.Checked ? 1 : 0;
65:                 if (sender==traditionalMailCheckBox)
66:                     check[6]=traditionalMailCheckBox.Checked ? 1 : 0;
67:             }
68:         }
69: }
```

在按钮事件的处理程序中，依次判断每一个单选按钮的状态，也就是是否被选中。根据单选按钮的状态，组织了一行文本在 Messages 中，最后通过一个 MessageBox 将其显示了出来。

有时需要将选择的状态存储下来供后续的处理使用。为此，用复选框来示例。首先在代码 50 行处声明了一个整数数组，用来记录复选框的状态。一共有 7 个复选框，哪一个复选框被选中，就将其对应的数组置为 1；若没选中则置为 0。复选框的状态改变时，会产生 CheckedChanged 事件，可以为每个复选框编写该事件的处理程序，改变 check 数组的值。那样会有 7 个差不多的事件处理程序，稍显凌乱。也可以这样，编写一个事件处理程序，7 个复选框共用。为此，首先添加第一个复选框的 CheckedChanged 事件处理代码，在上面的例子中，没有使用 Visual Studio 给出的方法名称，而是改为一个更一般的名称 CheckBox_CheckedChanged（代码 51 行）。在这个事件处理方法中，检测参数 sender 来

判定哪个复选框触发了该事件，随后对 check 数组做出相应的改变。为剩余的 6 个复选框添加 CheckedChanged 事件处理程序时，没有产生新的处理程序，而是单击下拉箭头，在已有的事件处理程序中选择一个。

11.2.4 图片框

众所周知，一幅图片中的信息量相当于上千个词，而实际上许多程序也都用图片来传递信息。此外在程序中，图片的使用还可以使程序更加生动。有若干方法可以显示图片，其中最常见的方法是使用图片框(PictureBox)控件。

PictureBox 控件可以显示来自位图、图标或者元文件，以及来自增强的元文件、JPEG 或 GIF 文件的图形。如果控件不足以显示整幅图像，则裁剪图像以适应控件的大小。

PictureBox 常用的属性有：

- Image：PictureBox 控件充当一个图片容器，可以通过设置 Image 属性来选择要显示的图片。Image 属性可以在"属性"窗口中设置，或者也可以编写代码，告诉程序显示哪幅图片。
- ImageLocation：要在图片框中显示的图片的路径或 URL。
- SizeMode：在 PictureBox 控件中伸展、居中对齐或者缩放图像。这是一个枚举值，可以是 Normal、StretchImage、AutoSize、CenterImage 和 Zoom。Normal(缺省值)将图像放在图片框的左上角，而 CenterImage 会将图像放在中间。如果图像比图片框大，则这两个选项只会显示部分图像。StretchImage 选项将图像调整为适合图片框的大小，AutoSize 将图片框的大小调整为能容纳图像，Zoom 将图像大小调整为适合图片框，但保持长宽比不变。

图片框的默认事件是 Click 事件。在设计模式双击 PictureBox 控件，会产生该事件的空白处理方法。

例 11-3 在本例中，在项目的目录中有 01.jpg、02.jpg 和 03.jpg 三个图像文件。使用图片框显示这 3 个文件，并且在单击图片框时，在 3 个图片之间轮流切换。

如果使用 Image 属性来设置图片，则首先需要将图片文件构建为一个 Bitmap 对象。而 Bitmap 是从 Image 类派生的，所以可以将 Bitmap 对象转换为父类 Image 对象。也可以直接使用 ImageLocation 属性，给出一个 string 型的路径。本例首先使用了该方法，然后示例了 bitmap。建立工程，将一个 PictureBox 控件放置到窗体上，设置 SizeMode 的属性为 Zoom。添加 Click 事件处理程序，代码如下：

```
using System;
using System.Windows.Forms;

namespace CSHARP11_3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```

```
10:                 InitializeComponent();
11:             }
12:             private int imageNum=-1;              //显示哪一个图片
13:             private void pictureBox1_Click(object sender,EventArgs e)
14:             {
15:                 imageNum=(imageNum+1)%3;
16:                 switch (imageNum)
17:                 {
18:                     case 0:
19:                         pictureBox1.ImageLocation="01.jpg";
20:                         break;
21:                     case 1:
22:                         pictureBox1.ImageLocation="02.jpg";
23:                         break;
24:                     case 2:
25:                         pictureBox1.ImageLocation="03.jpg";
26:                         break;
27:                 }
28:             }
29:     }
30: }
```

从代码的第 15 行到第 25 行，单击 PictureBox 时候，分别在 3 个图片之间轮流切换，显示效果如图 11-8 所示。

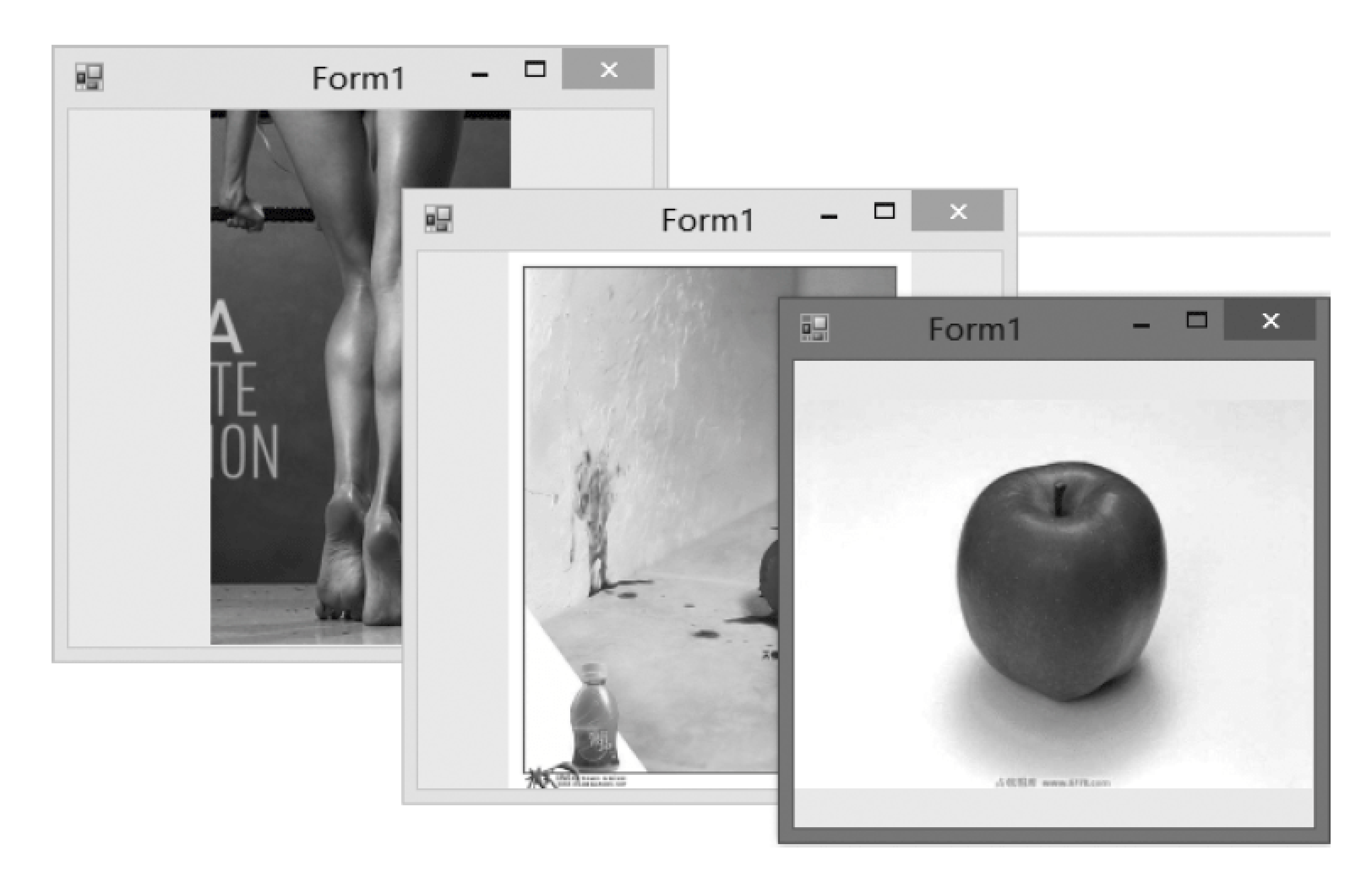


图 11-8　使用图片框显示图像

如果使用 bitmap 对象，则 Click 事件的代码如下：

```
private void pictureBox1_Click(object sender,EventArgs e)
{
    imageNum=(imageNum+1)%3;
```

```
    System.Drawing.Bitmap bitmap=null;
    switch (imageNum)
    {
        case 0:
            bitmap=new System.Drawing.Bitmap("01.jpg");
            break;
        case 1:
            bitmap=new System.Drawing.Bitmap("02.jpg");
            break;
        case 2:
            bitmap=new System.Drawing.Bitmap("03.jpg");
            break;
    }
    pictureBox1.Image=bitmap;
}
```

在上面的代码中，如果图片缺失，或者图片不在正确的目录中，则程序就无法正确显示图片。所以，在将图片添加到 PictureBox 控件之前，通常先将图片文件作为资源添加到项目中。一旦在项目中添加了某项资源，便可以根据需要任意重复使用该资源。例如，可以在多个位置显示同一图片，而不用担心程序移动到不同位置或计算机时的图像显示问题。

在创建一个工程后，可以用如下步骤将图像作为资源添加到工程中：

(1) 在解决方案资源管理器中，找到这个项目下的 Properties 结点，右击并选择打开，这时会打开该工程的属性页面。

(2) 选择左边标签中的资源标签。

(3) 在资源页的顶部单击添加资源按钮旁边的下拉箭头，选择添加已存在的文件。找到 3 个图片文件，依次添加进来，如图 11-9 所示。

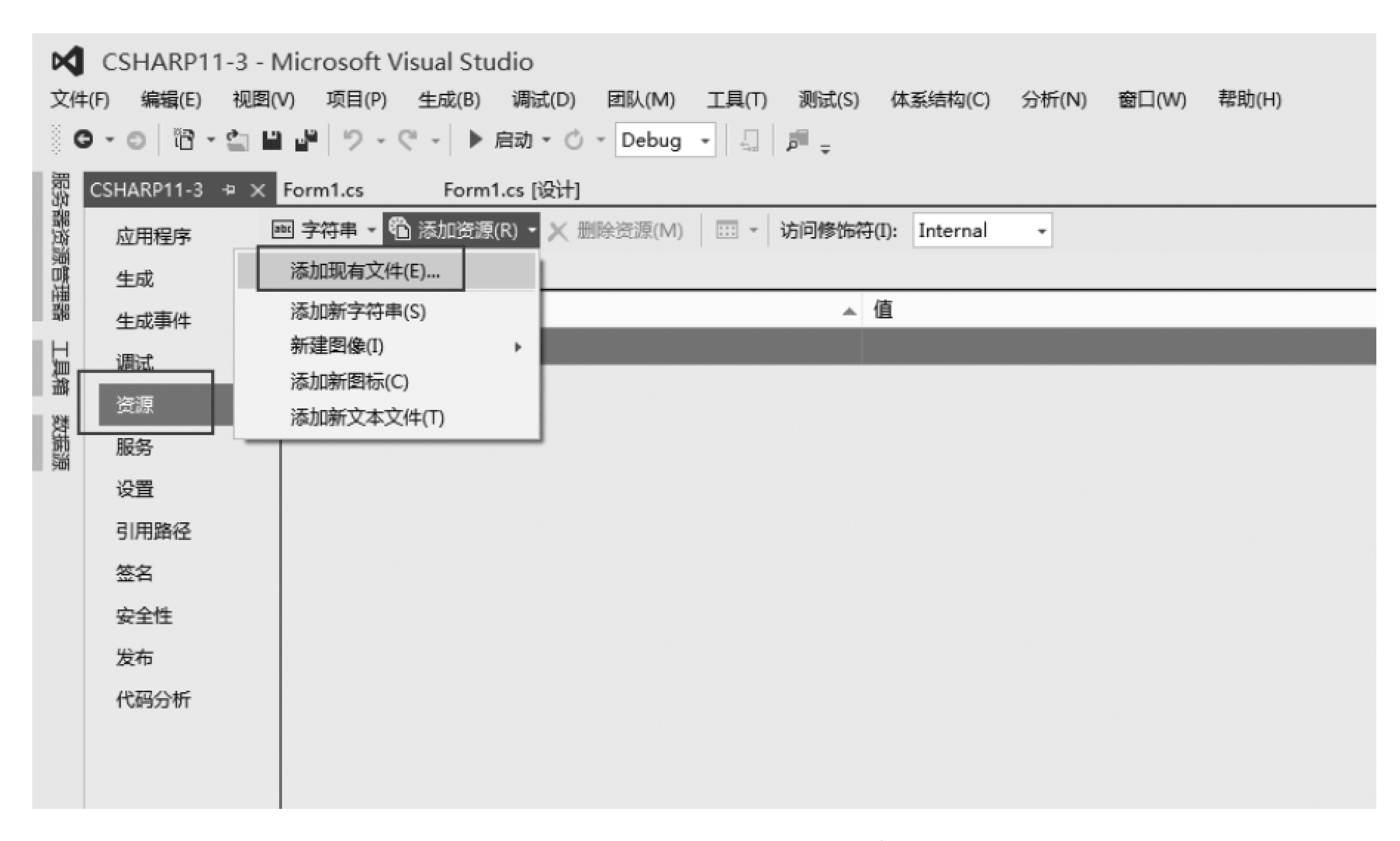


图 11-9　打开项目属性中的资源页

(4) 添加后的资源有一个名字,可以通过这个名字使用它(如图 11-10 所示)。

(5) 保存工程。

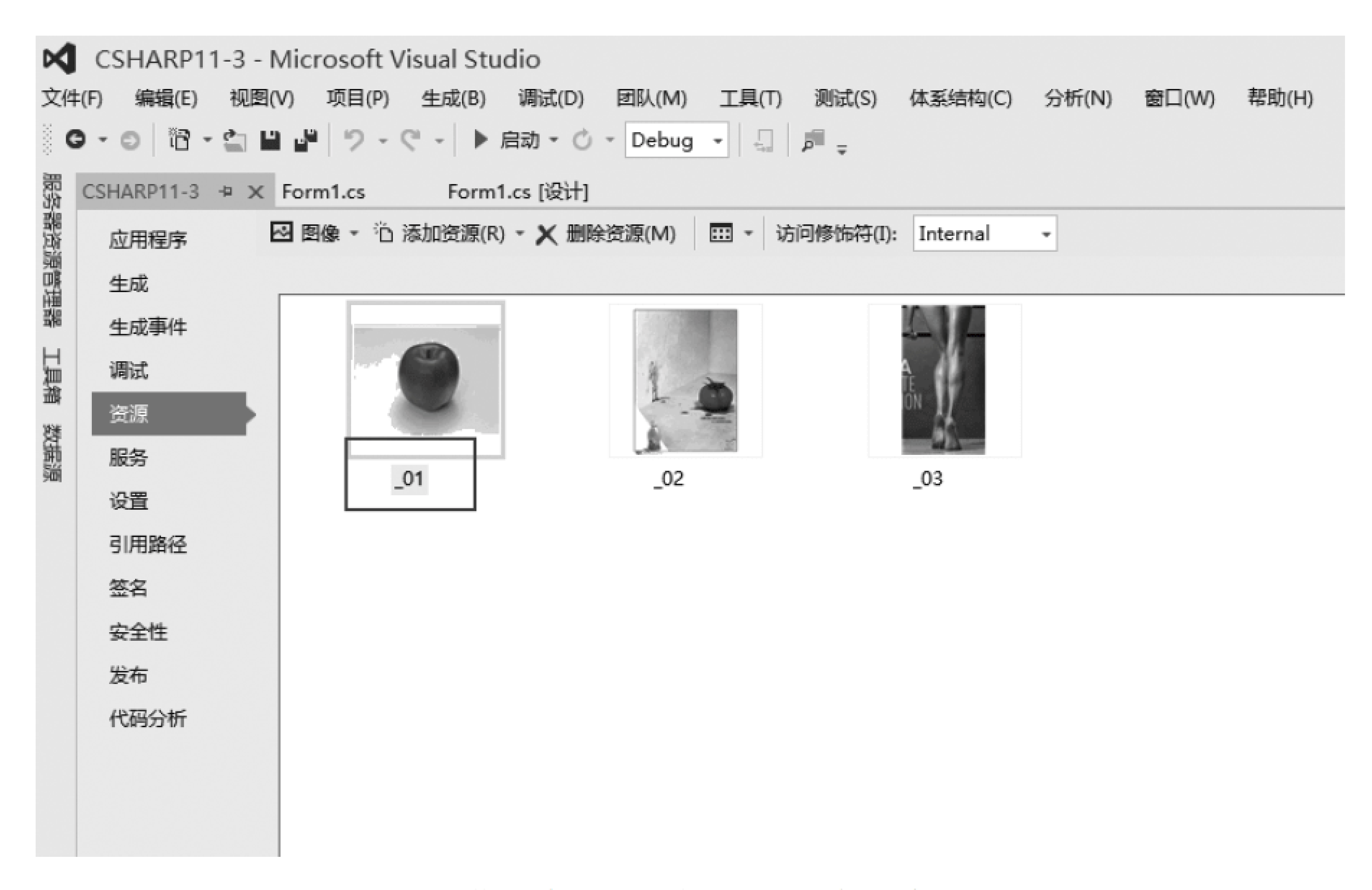


图 11-10 图片作为资源被添加进工程,每个资源有一个名字

添加后的资源本身就是一个 Bitmap 类型的对象,该对象可以 Properties 名字空间中的 Recourse 来应用,可以直接将该值赋给 PictureBox 的 Image 属性,并且也不用操心程序运行时图片在哪了,它已经嵌入到程序中了。此时,Click 事件的代码如下:

```
private void pictureBox1_Click(object sender,EventArgs e)
{
    imageNum= (imageNum+ 1) %3;
    switch(imageNum)
    {
        case 0:
            pictureBox1.Image= Properties.Resources._01;
            break;
        case 1:
            pictureBox1.Image= Properties.Resources._02;
            break;
        case 2:
            pictureBox1.Image= Properties.Resources._03;
            break;
    }
}
```

11.2.5 工具提示

工具提示(ToolTip)是当鼠标在某特定区域上停留时显示的一个矩形窗口。工具提

示窗口包含一些编程者想要显示的文本。在这点上，工具提示同状态栏的作用是一样的，所不同的是工具提示当单击或者远离指定区域的时候就会消失，一般最熟悉的应该是与工具栏相关联的工具提示，如 Word 的工具提示可以帮助使用者确定程序图标的用途。

ToolTip 类通常用来向用户提示控件的预期用途。例如，可以为接受名称的 TextBox 控件指定工具提示文本，同时指定要键入到控件中的名称的格式。除了提供提示外，还可使用 ToolTip 类提供运行时状态信息。例如，当用户将指针移动到显示 Internet 连接状态的 PictureBox 控件上时，可以使用 ToolTip 类显示连接速度和线路质量数据。

ToolTip 常用的属性有：

- AutomaticDelay：工具提示的自动延迟(毫秒)。
- AutoPopDelay：当指针在具有指定工具提示文本的控件内保持静止时，工具提示出现的时间(毫秒)。
- InitialDelay：工具提示显示之前必须悬停在控件上的时间(毫秒)。
- ReshowDelay：指针从一个控件移到另一控件时，必须经过多长时间才会出现后面的工具提示窗口(毫秒)。
- ShowAlways：指示是否显示工具提示窗口，甚至是在其父控件不活动的时候。

常见的事件有：

- Draw：当绘制工具提示并将 OwnerDraw 属性设置为 true，以及 IsBalloon 属性为 false 时发生。
- Popup：在工具提示最初显示之前发生。这是 ToolTip 类的默认事件。

例 11-4 使用 ToolTip 工具提示。在本示例中，在窗体上将放置两个按钮，当鼠标移动到按钮上时，出现不同的提示。

建立一个新的项目后，在窗口上放置两个按钮。然后将一个 ToolTip1 控件拖动到窗体上，ToolTip1 会出现在设计模式中的窗体下方，而不是在窗体上，如图 11-11 所示。

一旦有工具提示加入到窗体，回到按钮的属性窗口，就会出现一个名为“ToolTip1 上的 ToolTip”的新的属性。如果将这个 ToolTip 的 Name 改为“helpSaveToolTip”，则按钮属性窗口对应的新的属性为“helpSaveToolTip 上的 ToolTip”。要使得鼠标指向按钮时有提示，只需要单击该属性旁的下拉箭头，设置要提示的值即可。将第一个按钮的提示设置为“保存文件”，将第 2 个按钮的提示设置为“将文件另存为一个新文件”。本例题没有添加任何代码，程序的运行结果如图 11-12 所示。

11.2.6 数字调节控件

有时，希望将用户的输入范围限制在特定的数字范围内，这可以使用数字调节控件(NumericUpDown)来完成。NumericUpDown 控件看起来像是一个文本框与一对箭头的组合，用户可以单击箭头来调整值。该控件显示并设置选择列表中的单个数值。用户可以通过单击向上和向下按钮、按向上键和向下键或键入一个数字来增大和减小数字。单击向上键时，值沿最大值方向增加；单击向下键时，值沿最小值方向减少。一个典型的例子就是音乐播放器上的音量控件。NumericUpDown 类中常见的属性有：

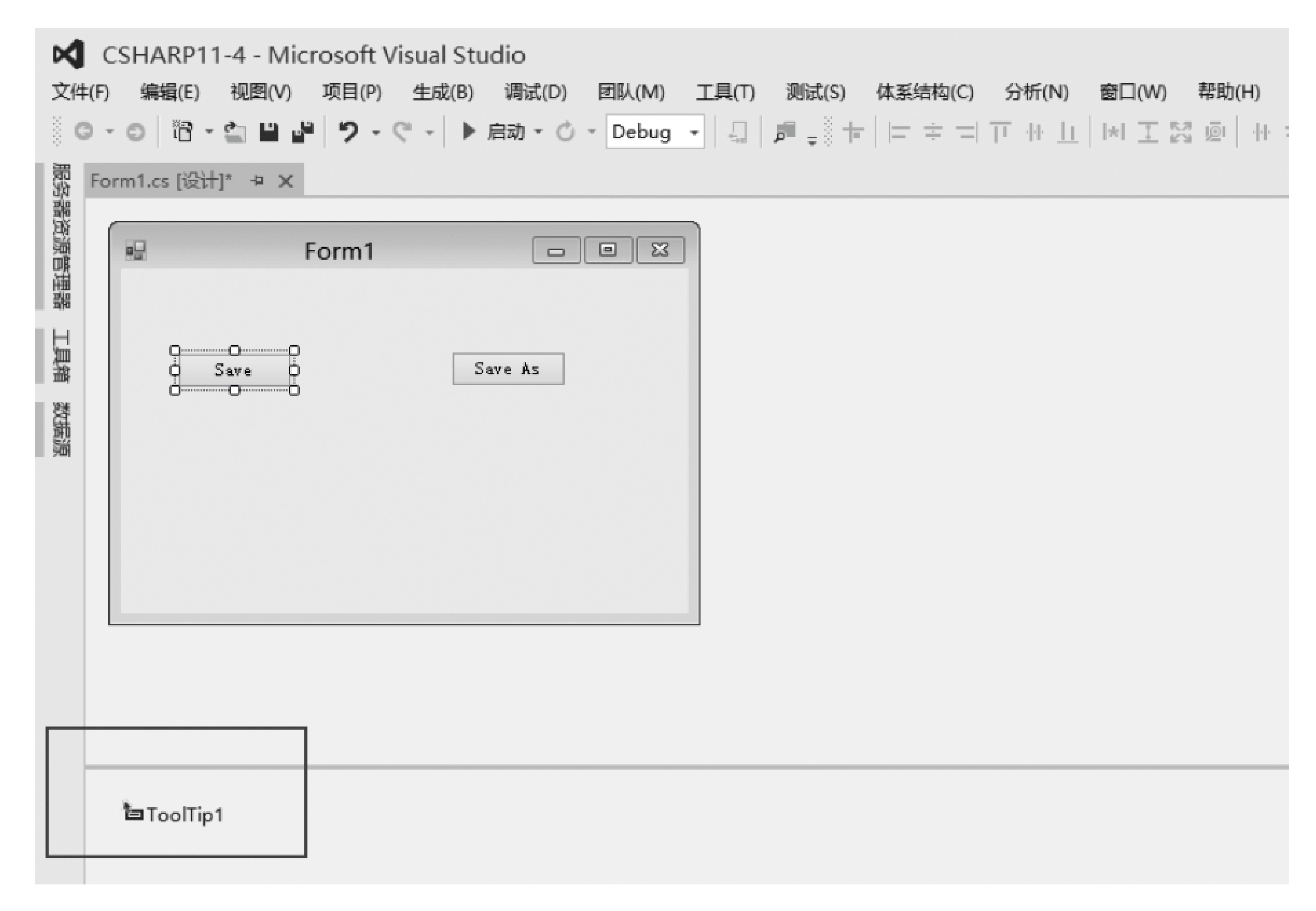


图 11-11　ToolTip 会出现在窗体的下方



图 11-12　ToolTip 程序的运行结果

- DecimalPlaces：NumericUpDown 控件中可以显示多少位小数。
- Increment：单击向上或向下按钮时，NumericUpDown 控件递增或递减的值。
- Maximum：NumericUpDown 控件的最大值。
- Minimum：NumericUpDown 控件的最小允许值。
- ReadOnly：指示是否只能使用向上或向下按钮更改文本。
- UpDownAlign：控件中向上和向下按钮的对齐方式。这个属性可用来将控件的这两个按钮显示在控件的左边或者右边。
- Value：控件的值。

控件的常见事件如下：

- ValueChanged：以某种方式更改 Value 属性后发生。

控件的数值类型是 decimal 类型，最大值和最小值用属性 Maximum 和 Minimum 指定。

例 11-5 数字调节控件的演示。在文本框中输入本金和利率，通过 NumericUpDown 控件输入年数，范围是 1～5，计算本息的和。程序的设计界面如图 11-13 所示。



图 11-13 程序的设计界面

将显示本息合计的文本框 ReadOnly 属性设置为 True；对于 NumericUpDown 控件，设置 Minimum＝1；Maximum＝5；同时，对各个控件重新命名。程序响应 ValueChanged 事件，NumericaUpDown 控件的值一发生改变，就立刻计算新值并显示。代码如下：

```
using System;
using System.Windows.Forms;

namespace CSHARP11_5
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void yearNumericUpDown_ValueChanged(object sender,
        EventArgs e)
        {
            //得到用户各项的输入
            decimal principal=Convert.ToDecimal(principalTextBox.Text);
            decimal rate=Convert.ToDecimal(interestTextBox.Text);
            int year=(int)yearNumericUpDown.Value;

            //计算本息合计
            decimal amount=principal *
            (decimal)Math.Pow((double)(1+rate/100),year);
            balanceTextBox.Text=amount.ToString();
        }
    }
}
```

11.2.7 Timer 组件

Timer 是定期引发事件的组件。Timer 只有两个重要的属性：

- Enabled 设置计时器是否启动。

- Interval 设置两次 Tick 事件的间隔。

时间间隔的长度由 Interval 属性定义，其值以毫秒为单位。若启用了该组件，则每个时间间隔引发一个 Tick 事件。Timer 组件的主要方法包括 Start 和 Stop，这两种方法可打开和关闭计时器（和直接设置 Enabled 属性的值的效果是一样的）。计时器在关闭时重置，不存在暂停 Timer 组件的方法。

11.3　鼠标事件处理

在程序的运行中，产生事件的主体很多。而程序的大部分输入是来自键盘或者鼠标的。当用户通过鼠标和控件交互时，就会产生关于鼠标的事件。从 Control 类派生的控件都可以处理鼠标事件。事件处理方法具有两个参数，第一个是 object 类型的对象，指出事件产生的主体，第二个参数则包含了事件的参数。对于鼠标事件而言，第二参数有两种类型：EventArgs 或者 MouseEventArgs。最主要的鼠标事件有以下 6 种，其中前 3 种的参数是 EventArgs，后 3 种的参数是 MouseEventArgs。

- MouseEnter：当鼠标进入控件的边界时产生此事件。
- MouseHover：当鼠标悬停在控件上时产生。
- MouseLeave：当鼠标离开控件时产生。
- MouseDown：当鼠标位于控件内，且按下鼠标按钮（无论哪一个）时产生。
- MouseUp：当鼠标位于控件内，松开鼠标按钮（无论哪一个）时产生。
- MouseMove：当鼠标位于控件内，并且移动鼠标时产生。

MouseEventArgs 的主要成员有：

- Button：按下的是哪个鼠标按钮。
- Clicks：按下并释放鼠标按钮的次数。
- Delta：鼠标轮已转动的制动器数（鼠标轮的齿轮）的有符号计数。当旋转鼠标轮时，每碰到一个齿就会发送一个鼠标轮消息。Windows 常数 WHEEL_DELTA 定义了一个鼠标轮总齿数，标准值为 120。正值指示鼠标轮向前（远离用户的方向）转动，负值指示鼠标轮向后（朝着用户的方向）转动。
- Location：鼠标在产生鼠标事件时的位置。
- X：鼠标在产生鼠标事件时的 x 坐标。
- Y：鼠标在产生鼠标事件时的 y 坐标。

例 11-6　在本例中演示鼠标事件。为了较为清楚地看到鼠标事件的发生，在此随着鼠标的移动在窗体上绘出轨迹。有关绘画部分请参阅本书后边的章节。本例是演示鼠标事件的，所以绘图代码直接放到了鼠标事件处理中，更好的方式是在 Paint 事件中处理。程序的界面设计如图 11-14 所示。

首先在窗体上放置一个横贯整个窗体的 Panel 控件。为了看清楚这个控件，设置其 BorderStyles 属性为 FixedSingle。在这个 Panel 控件上接着放置了 4 个大小一样的 Panel，并将左边的 3 个 Panel 的 BackColor 属性分别设置为 Red、Green 和 Blue。当鼠标进入某种颜色的 Panel 后，绘图的画笔就将保持该颜色，直到被新的颜色替换。

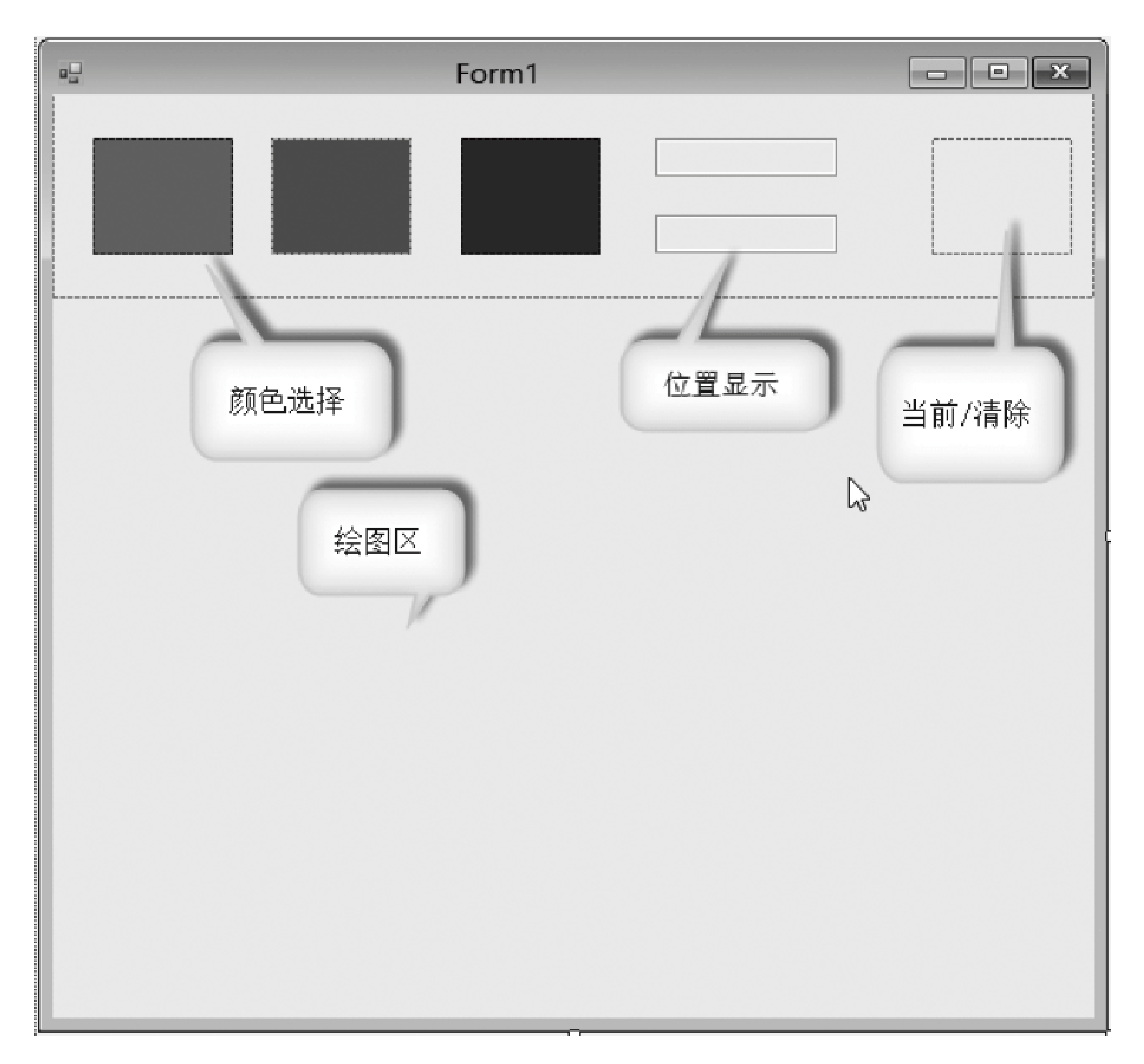


图 11-14　程序的界面设计

画笔一开始的颜色和窗体的背景色一样，所以不选颜色就画会看不到画出的轨迹。选择某个颜色后，最右边的 Panel 会显示当前选择的颜色。当鼠标在最右边的 Panel 上悬停时，会清除掉所有的选择和绘画。为了使这个 Panel 看得清楚些，设置其 BorderStyle 属性为 Fixed3D。Panel 中间的两个文本框显示鼠标在窗体的 X 坐标和 Y 坐标，当鼠标离开绘画区后将保持坐标为最后的值不变。

当按住鼠标左键(不放)并在窗口上移动鼠标时，用当前选择的颜色绘制出鼠标的轨迹。释放鼠标按钮后，将不再绘制轨迹。设置两个文本框的 ReadOnly 属性为 True，整个程序的代码如下：

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace CSHARP11_6
{
    public partial class Form1 : Form
    {
        private Color pickedColor;              //当前选择的颜色
        private Point old;                      //鼠标上一个点的坐标
        private bool capture= false;            //鼠标键是否按下
        public Form1()
```

```
        {
            InitializeComponent();
            pickedColor=this.BackColor;          //初始颜色和窗体背景色一致
        }

        private void redPanel_MouseEnter(object sender,EventArgs e)
        {
            pickedColor=Color.Red;
            pickedColorPanel.BackColor=pickedColor;
        }

        private void greenPanel_MouseEnter(object sender,EventArgs e)
        {
            pickedColor=Color.Green;
            pickedColorPanel.BackColor=pickedColor;
        }

        private void bluePanel_MouseEnter(object sender,EventArgs e)
        {
            pickedColor=Color.Blue;
            pickedColorPanel.BackColor=pickedColor;
        }

        private void pickedColorPanel_MouseHover(object sender,EventArgs e)
        {
            pickedColor=this.BackColor;
            pickedColorPanel.BackColor=pickedColor;
            using (Graphics g=CreateGraphics())
            {
                g.Clear(pickedColor);
            }
        }

        private void Form1_MouseDown(object sender,MouseEventArgs e)
        {
            if(e.Button==MouseButtons.Left)
            {
                capture=true;
                old=e.Location;
            }
        }

        private void Form1_MouseUp(object sender,MouseEventArgs e)
        {
```

```
57:                if (e.Button==MouseButtons.Left)
58:                {
59:                    capture=false;
60:                }
61:            }
62:
63:            private void Form1_MouseMove(object sender,MouseEventArgs e)
64:            {
65:                if(capture)
66:                {
67:                    using(Graphics g=CreateGraphics())
68:                    {
69:                        Pen pen=new Pen(pickedColor);
70:                        g.DrawLine(pen,old,e.Location);
71:                        old=e.Location;
72:                    }
73:                }
74:                xValueTextBox.Text=e.Location.X.ToString();
75:                yValueTextBox.Text=e.Location.Y.ToString();
76:            }
77:        }
78: }
```

代码 18～34 行分别是鼠标进入 3 个颜色面板的事件处理代码，在这些代码中将最右边一个面板的背景色设置为与鼠标进入的面板背景色一致，同时将该颜色存储在 pickedColor 中。变量 capture（第 11 行）判断是否绘图。当左键被按下时（代码 46～53 行），capture 设置为 ture，并记录按下的位置。当左键释放时，capture 为 false（代码 55～61 行）。

在鼠标移动的事件处理代码中（代码 63～77 行），使用了一个绘图对象（后面章节讲述）用直线连接鼠标移动的每一点，来绘出鼠标移动的轨迹。同时将鼠标的位置写入到文本框中。在鼠标悬停在最右边的 Panel 上时，清除了窗体上的绘图。using 语句（第 67 行）表示使用完 Graphic 对象后自动调用 Dispose 方法释放该对象。

11.4 键盘事件处理

和鼠标事件一样，凡是从 Control 类派生的控件都可以处理键盘事件。键盘事件在按下和释放键盘上的键时发生。键盘事件有以下 3 个：

- KeyDown：按下键时产生。
- KeyUp：释放（松开）键时产生。
- KeyPress：按下键时产生，出现在 KeyDown 事件之后、KeyUp 事件之前。

KeyPress 事件处理方法使用类 KeyPressEventArgs 传递事件参数。在这个类中只有两个属性：

- Handle：指示该事件是否被处理过，如果未被处理，则会将它发送到操作系统进行默认处理。将 Handled 设置为 true，则可以取消 KeyPress 事件。
- KeyChar：返回按下键的 ASCII 字符。

KeyDown 和 KeyUp 消息使用类 KeyEventArgs 传递消息参数，这个类的主要属性如下：

- Alt：指示是否按下 Alt 键。
- Control：指示是否按下 Ctrl 键。
- Handled：指示是否处理过此事件。
- KeyCode：获取 KeyDown 或 KeyUp 事件的键盘代码。
- KeyData：获取 KeyDown 或 KeyUp 事件的键数据。
- KeyValue：获取 KeyDown 或 KeyUp 事件的键盘值。
- Modifiers：获取 KeyDown 或 KeyUp 事件的修饰符标志。这些标志指示按下的 Ctrl、Shift 和 Alt 键的组合。
- Shift：指示是否按下 Shift 键。

KeyDown 和 KeyPress 的主要区别：KeyPress 主要用来接收字母、数字等字符，KeyPress 只能捕获单个字符。KeyPress 不显示键盘的物理状态(Shift 键)，而只是传递一个字符。KeyPress 将每个字符的大、小写形式作为不同的键代码解释，即作为两种不同的字符。KeyPress 不区分小键盘和主键盘的数字字符。

KeyDown 和 KeyUP 事件过程通常可以捕获键盘除了 PrScrn(在键盘右上角)的所有按键。KeyDown 和 KeyUp 可以捕获组合键。KeyDown 和 KeyUp 不能判断键值字母的大小，返回的总是大写字母。KeyDown 和 KeyUp 区分小键盘和主键盘的数字字符。

在 KeyDown 事件中，KeyEventArgs 有 3 个主要的属性传递键值，主要区别是：KeyCode 最为常用，记录在键盘上按了哪个键，当使用组合键时，如 Ctrl＋A，其值是"A"；KeyData 可以记录组合键，当使用组合键时，如 Ctrl＋A，其值为"A，Ctrl"；KeyValue 则是 KeyCode 的数字值，当使用组合键时，如 Ctrl＋A，其值为 65(A)，注意不是 97(a)。

例 11-7　键盘按键测试。建立一个新项目。对窗体的 3 个键盘事件编写代码并测试。测试的结果使用 MessageBox 来显示，代码如下：

```
using System.Windows.Forms;

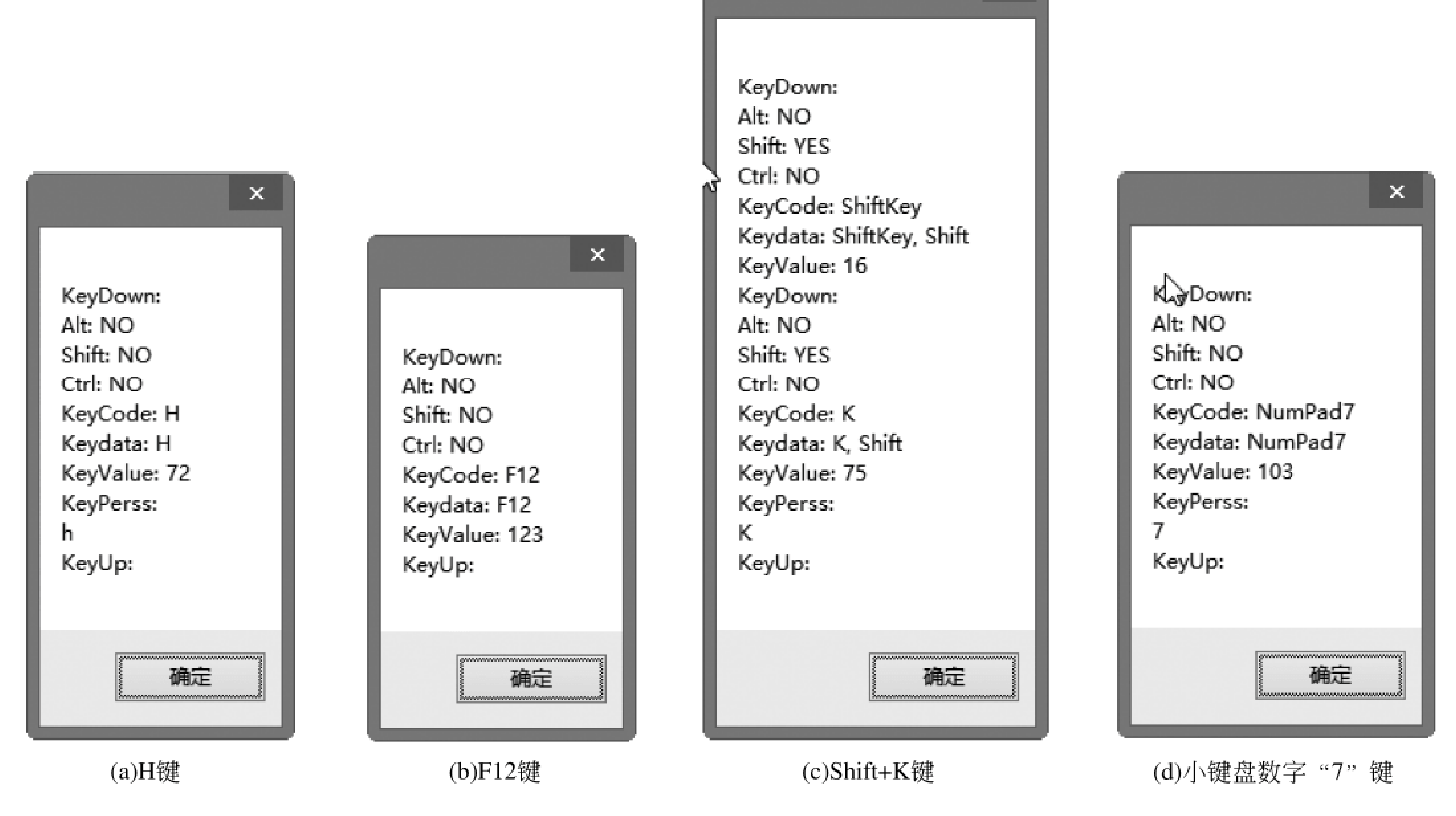
namespace CSHARP11_7
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private string eventMsg;                              //要显示的内容
        private void Form1_KeyDown(object sender,KeyEventArgs e)
        {
```

```
                eventMsg+ = "KeyDown: \n"+
                    "Alt: "+ (e.Alt ? "Yes" : "NO")+ '\n'+
                    "Shift: "+ (e.Shift ? "YES" : "NO")+ '\n'+
                    "Ctrl: "+ (e.Control ? "Yes" : "NO")+ '\n'+
                    "KeyCode: "+ e.KeyCode+ '\n'+
                    "Keydata: "+ e.KeyData+ '\n'+
                    "KeyValue: "+ e.KeyValue+ '\n';
            }

            private void Form1_KeyPress(object sender,KeyPressEventArgs e)
            {
                eventMsg+ = "KeyPerss: \n"+ e.KeyChar+ '\n';
            }

            private void Form1_KeyUp(object sender,KeyEventArgs e)
            {
                eventMsg+ = "KeyUp: \n";
                MessageBox.Show(eventMsg);
                eventMsg= "";
            }
        }
}
```

在测试中,分别按下了 H 键、F12 键、Shift+K 键和小键盘的数字"7"键,显示结果如图 11-15 所示。



(a)H键 (b)F12键 (c)Shift+K键 (d)小键盘数字"7"键

图 11-15 显示结果

按下 H 键并松开的时候,会依次激发 KeyDown、KeyPress 和 KeyUp 事件。在 KeyDown 的事件中,参数传递的是键 H,不区分大小写;而在 KeyPress 中传递的是小写的 h。按下 F12 键时,因为没有对应的字符,所以不会产生 KeyPress 事件。按下 Shift+K 键时,KeyDown 事件发送了 2 次,Shift 键按下和 K 键按下。KeyPress 事件得到了大写的 K。按下小键盘键时,KeyPerss 发生,但不区分是小键盘的"7"还是主键盘上的"7";而 KeyDown 事件是区分的,传递的是 NumPad7。

11.5 常用控件 2

11.5.1 MonthCalendar 和 DateTimePicker 控件

MonthCalendar 和 DateTimePicker 控件都是用来选取时间的。Windows 窗体 MonthCalendar 控件为用户查看和设置日期信息提供了一个直观的图形界面。该控件以网格形式显示日历,网格包含月份的编号日期。这些日期排列在周一到周日的七个列中,并且突出显示选定的日期范围。可以单击月份标题任何一侧的箭头来选择不同的月份。与类似的 DateTimePicker 控件不同,可以使用该控件选择多个日期。

MonthCalendar 控件的外观具有很高的可配置性。默认情况下,今天的日期显示为圆形,并且在网格的底部加以说明。将 ShowToday 和 ShowTodayCircle 属性设置为 false,可以更改此功能。还可以通过将 ShowWeekNumbers 属性设置为 true,在日历中添加周编号。通过设置 CalendarDimensions 属性,可以水平和垂直显示多个月份。默认情况下,星期日显示为每周的第一天,不过可以使用 FirstDayOfWeek 属性将任何一天指定为第一天。

此外,还可以通过向 BoldedDates、AnnuallyBoldedDates 和 MonthlyBoldedDates 属性添加 DateTime 对象,将某些日期设置为一次性地、每年或每月显示为粗体。MonthCalendar 控件的主要属性是 SelectionRange,即该控件中选定的日期范围。SelectionRange 值不能超过 MaxSelectionCount 属性中设置的最大可选择天数。用户可以选择的最早和最晚日期由 MaxDate 和 MinDate 属性确定。

使用 Windows 窗体 DateTimePicker 控件,用户可以从日期或时间列表中选择单个项。在用来表示日期时,它显示为两部分:一个下拉列表(带有以文本形式表示的日期)和一个网格(在单击列表旁边的向下箭头时显示)。该网格看起来很像可用于选择多个日期的 MonthCalendar 控件。

DateTimePicker 作为选取或编辑时间(而不是日期)的控件出现,将 ShowUpDown 属性设置为 true,并将 Format 属性设置为 Time。当 ShowCheckBox 属性设置为 true 时,该控件中的选定日期旁边将显示一个复选框。当选中该复选框时,选定的日期时间值可以更新。当复选框为空时,值显示为不可用。

该控件的 MaxDate 和 MinDate 属性确定日期和时间的范围。Value 属性包含该控件设置的当前日期和时间。值可以按以下四种格式显示(这些格式通过 Format 属性设置):Long、Short、Time 或 Custom。如果选择自定义格式,则必须将 CustomFormat 属

性设置为适当的字符串。

例 11-8 DateTimePicker 的简单示例。从窗体的 DateTimePicker 控件中选择一个日期，然后在下方的文本框中显示 5 个工作日后是哪一天。代码如下：

```
using System;
using System.Windows.Forms;

namespace CSHARP11_8
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

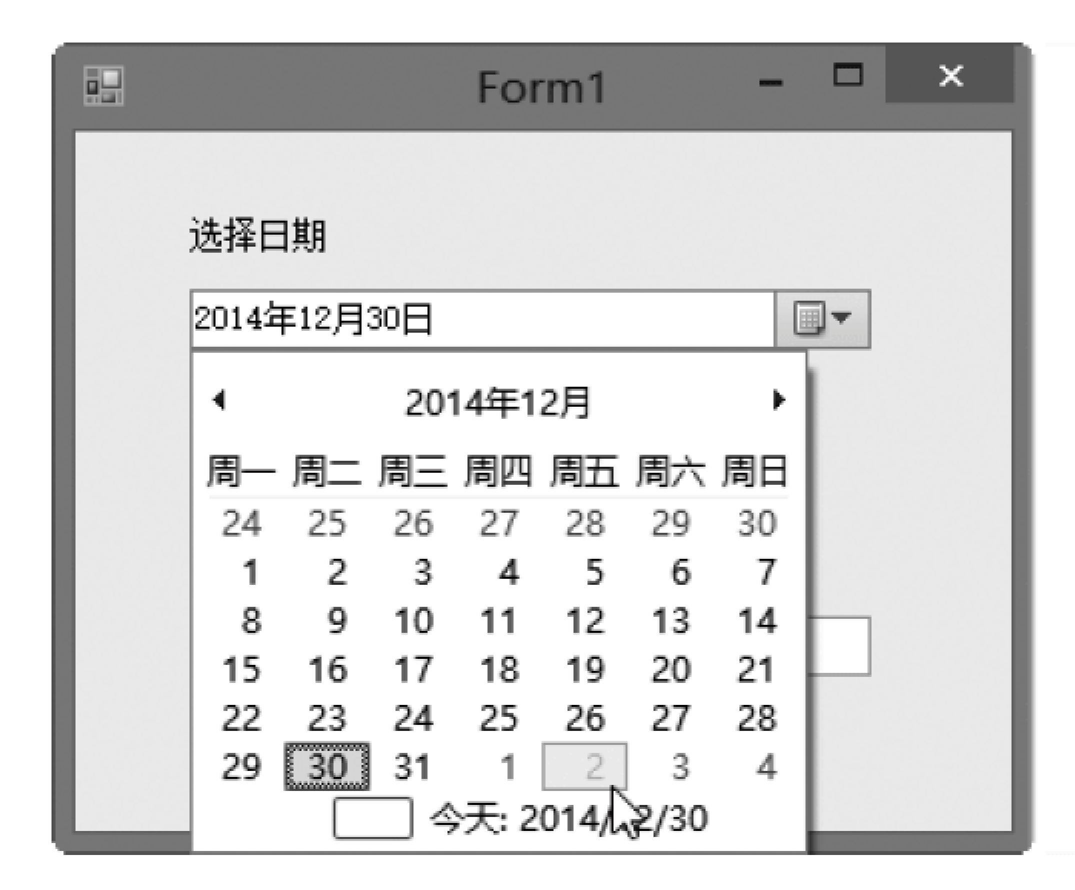
        private void dateTimePicker1_ValueChanged(object sender,
                EventArgs e)
        {
            DateTime pickedDate= dateTimePicker1.Value;

            //向后加 5 天
            int j= 0;
            DateTime fiveDays= pickedDate;
            while (j< 5)
            {
                fiveDays= fiveDays.AddDays(1);
                //跳过周六和周日
                if (fiveDays.DayOfWeek != DayOfWeek.Saturday &&
                    fiveDays.DayOfWeek != DayOfWeek.Sunday)
                    j++;
            }
            textBox1.Text= fiveDays.ToLongDateString();
        }
    }
}
```

程序运行结果如图 11-16 所示。

11.5.2 ListBox、CheckedListBox 和 ComboBox

RadioButton 和 CheckedBox 可以显示多个选项供用户选择，但每个单选按钮或检查框都要占据窗体上的空间。窗体上单选按钮或检查框出现得越多，选项也就越混乱。此时，可以使用 ListBox（列表框）、CheckedListBox（复选列表框）和 ComboBox（组合列表



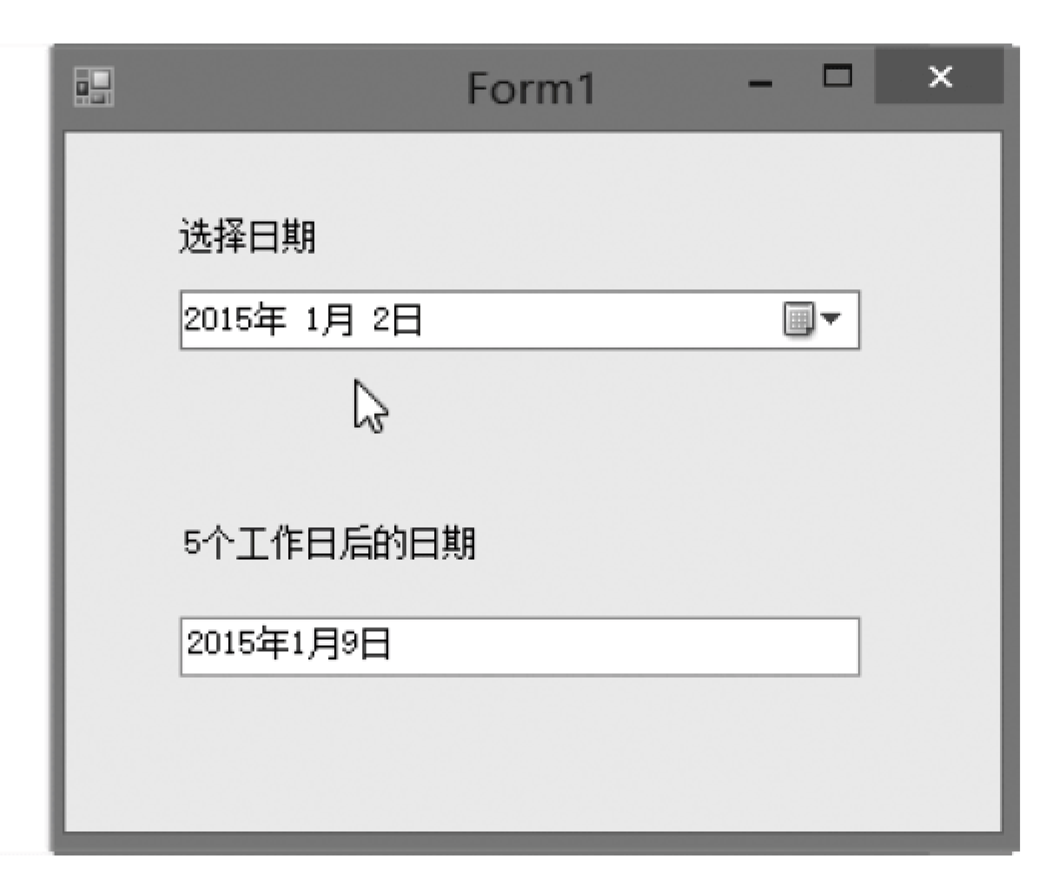


图 11-16 DateTimePicker 的简单示例

框)向用户提供选项。它们均可显示多个选项,而且比多个单选按钮或检查框占据的空间要少一些。

ListBox 控件用来显示选项列表,用户可从中进行选择一项或多项操作。在窗体创建一个列表框时,单击 ListBox 小图标。鼠标箭头变为十字形状,将鼠标移至窗体上适当位置,按住鼠标左键,拖动鼠标绘制列表框。

要在设计的时候为 ListBox 添加选项,可以使用属性窗口中的 Items 属性(位于 Data 类中)。打开 ListBox 的属性窗口,单击 Items 属性项右边的带有三个小黑点的按钮,Visual Studio 2008 将打开一个字符串集合编辑器。将相应的选项内容输入到字符串集合编辑器中。每输入一个选项,按回车键换行。全部选项输入完毕,单击 OK 按钮结束,如图 11-17 所示的 ListBox。如果项总数超出可以显示的项数,则自动向 ListBox 控件添加滚动条。

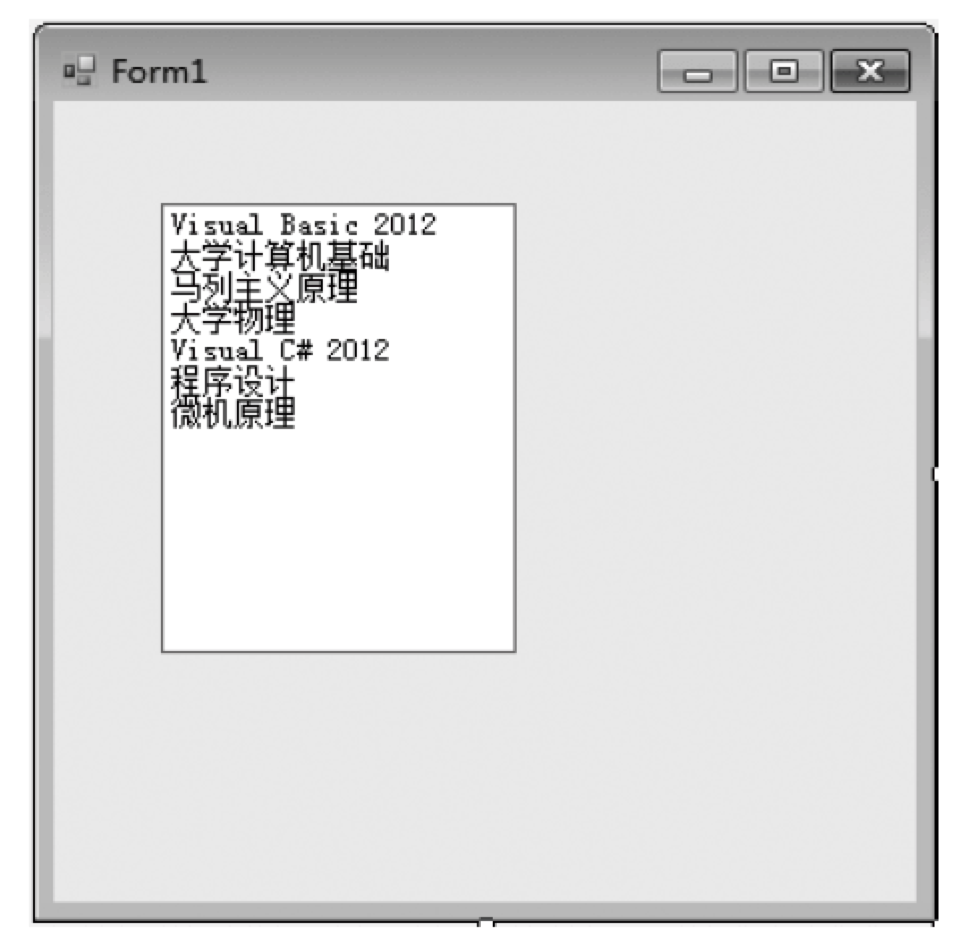


图 11-17 ListBox 控件

ListBox 的常用属性如下:

- SelectionMode 属性:该属性可以让用户在列表中选择多个项,有 4 个可选的值:One、None、MultiSimple 和 MultiExtend。分别表示如下含义:
  - One:一次只能选择一项,不接受多项选择。此时如果再选择第二项,则前一项的选定被取消。
  - None:用户不能在列表框中选择项。
  - MultiSimple:鼠标单击或按空格键将选择或撤消选择列表中的某项。
  - MultiExtended:按下 Shift 键的同时单击鼠标或者同时按 Shift 键和箭头键之一(上箭头键、下箭头键、左箭头键和右箭头键),会将选定内容从前一选定项扩展到当前项。按 Ctrl 键的同时单击鼠标将选择或撤消选择列表中的某项。

- SelectedIndex 属性：获取或设置 ListBox 中当前选定项从零开始的索引。该属性只能在程序中设置或引用。如图 11-16 所示，若用户选择了第四项“大学物理”，则 SelectedIndex 的值为 3(从 0 开始)。如果 ListBox 的 SelectionMode 属性设置为 MultiSimple 或 MultiExtended(它指示多重选择 ListBox)，并在该列表中选定多个项，则此属性返回任意一个选定项的索引。因此，若要检索包含多重选择 ListBox 中所有选定项的索引的集合，则需要使用 SelectedIndices 属性。如果没有选定项，则 SelectedIndex 为－1。
- SelectedItem 属性：获取或设置 ListBox 中的当前选定项。SelectedIndex 返回索引，而 SelectedItem 返回项中实际的内容。例如当 SelectedIndex 的值为 3 时，SelectedItem 的值为“大学物理”。同样，若要检索包含多重选择 ListBox 中所有选定项的集合，则需使用 SelectedItems 属性。
- SelectedIndices 属性：获取一个集合，该集合包含 ListBox 中所有当前选定项的从零开始的索引(集合的概念在数组一章中讲述)。
- SelectedItems 属性：对于多重选择 ListBox，此属性返回一个集合，该集合包含 ListBox 中选定的所有项。
- Sorted 属性：该属性决定列表框中的项目在程序运行期间是否按字母顺序排列显示。当 Sorted 值为 True 时，项目按字母顺序排列显示；为 False 时，项目按加入的先后顺序排列显示。该属性只能在设计状态时设置。
- Text 属性：该属性值是被选中的列表项的内容，类似于 SelectedItem。该属性只能在程序中设置或引用。

ListBox 的 Item 属性本身是一个集合对象，可以利用 Item 的方法在程序运行期间为 ListBox 添加新的项目或删除某一项。可以用 Add 方法在 ListBox 后添加一项：

```
ListBoxName.Item.Add("新的表项");
```

要将某一项插入到指定位置，可以使用 Insert 方法：

```
ListBoxName.Item.Insert(n,"新的表项");
```

其中 n 表示要插入的位置。需要删除一项时，可以使用 Remove 方法：

```
ListBoxName.Item.RemoveAt(n);
ListBoxName.Item.Remove("要删除的项");
```

要使用项目的索引值，可使用 RemoveAt；用 Remove 方法可通过删除项目本身的值来删除它。还可以使用 Clear 方法一次将所有项全部删除。

例 11-9 在 ListBox 中选中一项，通过单击删除按钮将它删去；选中的项的内容也同时复制到一个文本框中。也可以在 TextBox 中输入新项，单击加入按钮将此项添加到 ListBox 中。建立一个新的项目，对控件重新命名，编写两个按钮的 Click 事件方法以及 ListBox 的 SelectedIndexChanged 事件方法，代码如下：

```
01: using System;
02: using System.Windows.Forms;
```

```

namespace CSHARP11_9
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void delButton_Click(object sender,EventArgs e)
        {
            if (courseListBox.SelectedIndex >= 0)
                courseListBox.Items.RemoveAt(courseListBox.SelectedIndex);
        }

        private void addButton_Click(object sender,EventArgs e)
        {
            if (addTextBox.Text != "")
                courseListBox.Items.Add(addTextBox.Text);
        }

        private void courseListBox_SelectedIndexChanged(object sender,EventArgs e)
        {
            if (courseListBox.SelectedIndex>= 0)
            {
                selectedTextBox.Text= courseListBox.
                Items[courseListBox.SelectedIndex].ToString();
            }
            else
            {
                selectedTextBox.Text= "";
            }
        }
    }
}
```

程序的运行结果及界面如图 11-18 所示。

CheckedListBox(复选列表框)的使用本质上与 ListBox 是一样的,只是选项在列表中显示的样式稍有不同,即在每个选项前有一个方框,当要选该项时,单击前面的方框即可。图 11-19 是一个 CheckedListBox 控件的示例。

由图 11-19 可以看到,在每一项的前面有一个方框。方框里面有对号的表示已经选中的项。另外还有一项有一个蓝色的条表示目前突出显示的项(该图中是“大学计算机基础”项)。需要注意的是属性 SelectedItems 和 SelectedIndices 不确定哪些项已选中,而是

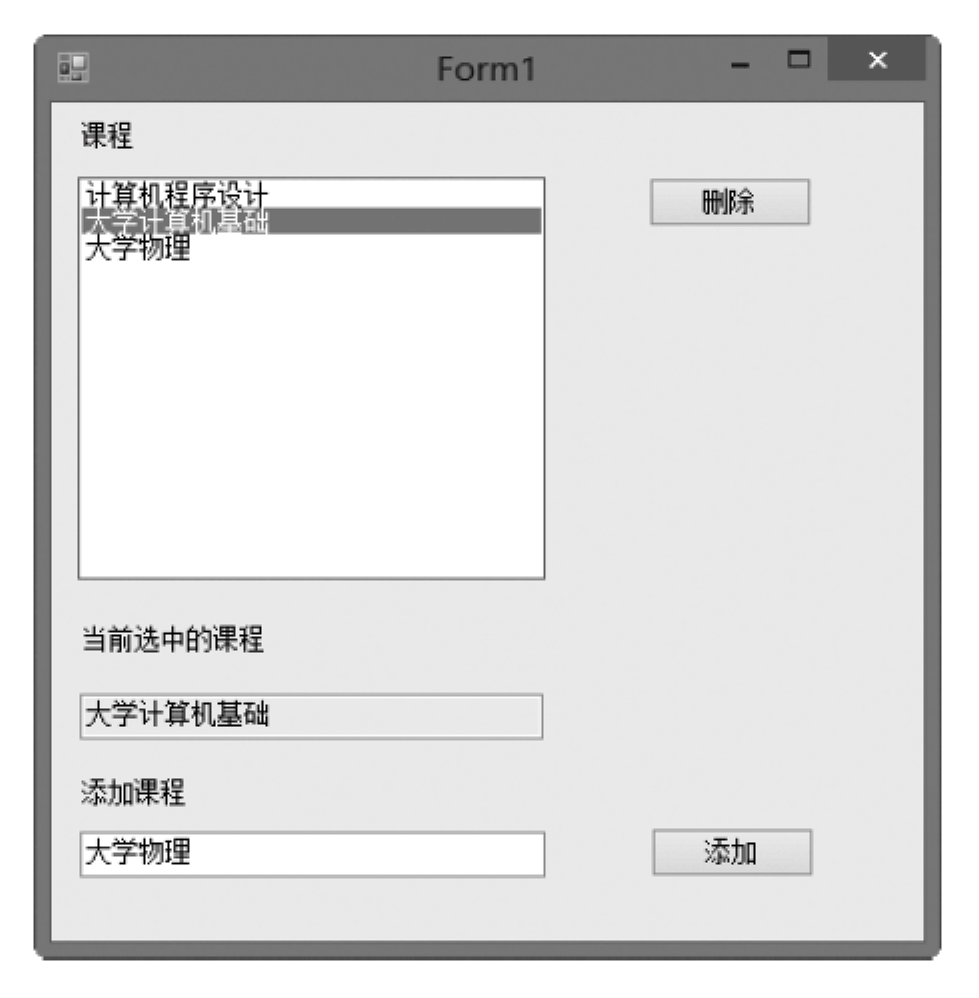



图 11-18　列表框的简单示例



图 11-19　CheckedListBox 控件

确定哪些项为突出显示的项。如果要确定哪些项被选中，则需要对每一项分别调用方法 GetItemChecked。若调用结果为 True，则表示被选中，否则未被选中。调用时，采用索引号指定某一项。对图 11-19 的 CheckedListBox，若有：

```
CheckListBoxName.GetItemChecked(0)
CheckListBoxName.GetItemChecked(2)
```

则由于第一项被选中，所以第一个式子的结果为 True；而第二项未被选中，因此结果为 False。CheeckedListBox 控件的添加项、删除项等方法与 ListBox 是一样的。

ComboBox（组合框）控件结合了文本框和列表框二者的特点。ComboBox 控件用于在下拉组合框中显示数据。默认情况下，ComboBox 控件分两个部分显示。顶部是一个允许用户键入列表项的文本框。第二个部分是列表框，它显示用户可以从中选择的项的列表。组合框在列表框中列出可供用户选择的项。当列表框中没有所需的选项时，允许用户在文本框中输入一个新的项，但输入的内容不能自动添加到列表框中。若用户选中列表框中的某项，则该项内容会自动装入文本框中。默认的 ComboBox 控件如图 11-20 所示。

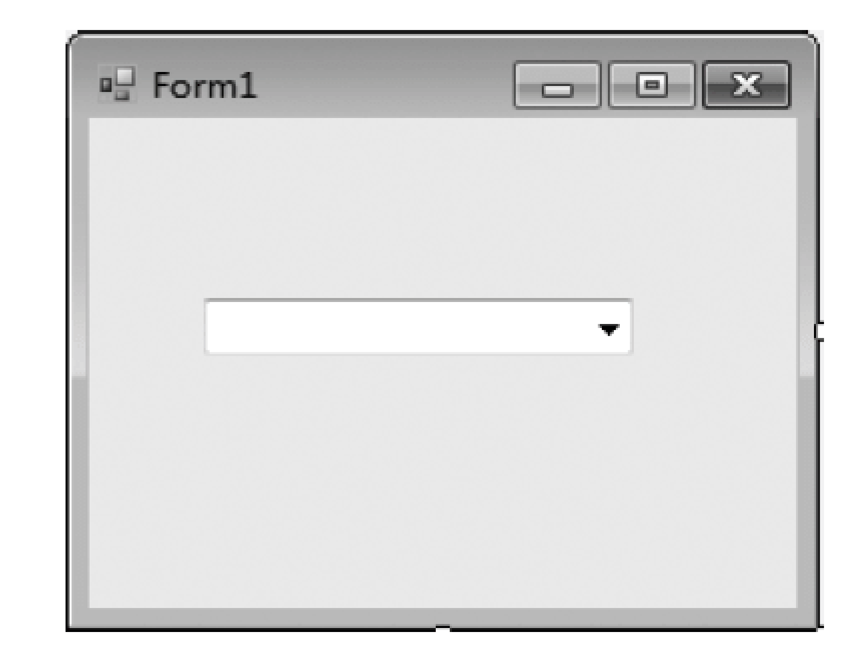


图 11-20　默认状态下的 ComboBox 控件

和 ListBox 控件一样，可以在 Items 属性中输入选项。程序运行后可以单击 ComboBox 控件的下拉箭头看到供选择的项，如图 11-21 所示。

可以通过设置 DropDownStyle 属性使得 CheckedListBox 控件有 3 种不同的外观和运行方式，分别是 DropDown、Simple 和 DropDownList，如图 11-22 所示。

左边最上方是 DropDown 模式，前面使用的都是这种模式。已经知道通过单击下拉箭头可以作出选择，也可以在文本框种输入新的项。下方是 DropDownList 模式，和

DropDown 模式的区别在于用户只能选择，不能输入新的项。右边是 Simple 模式，没有下拉箭头，直接将选项显示了出来。很明显，这种方式比较占空间。在其他方面，它和 ListBox 控件的用法基本相同。

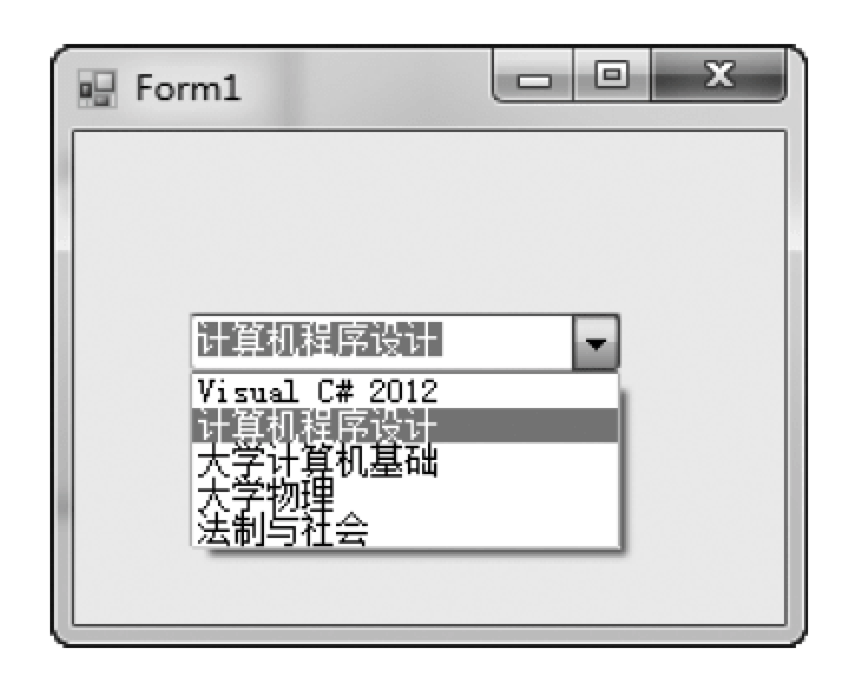


图 11-21　程序运行时通过单击下拉箭头来选择需要的项

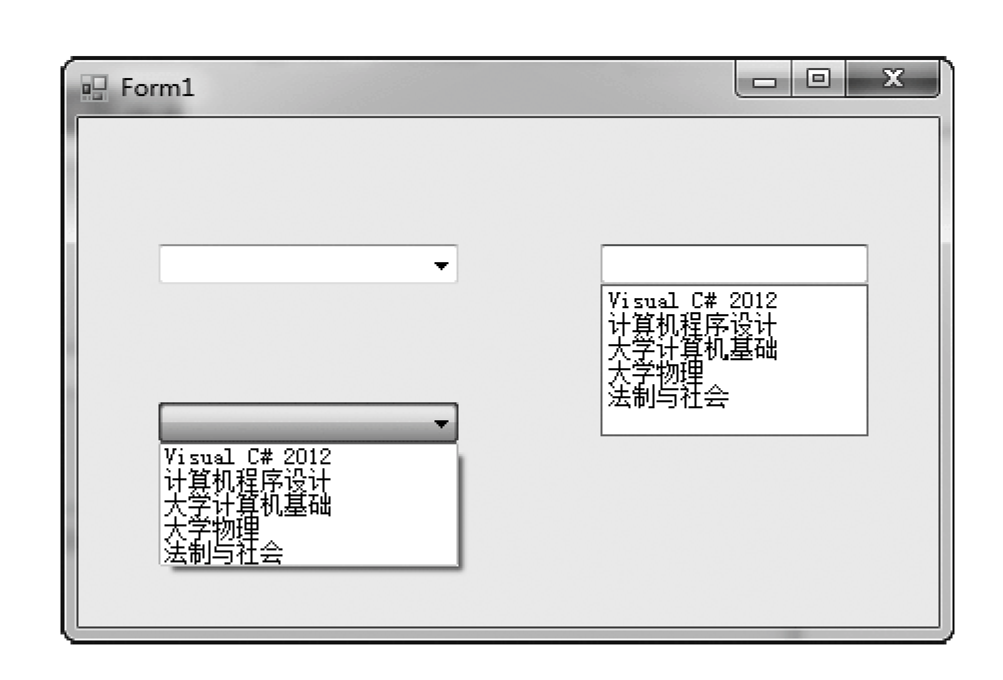


图 11-22　3 种模式

11.5.3　TreeView 和 ListView

TreeView 控件用树显示层次结点。传统上，结点对象包含值，可以引用其他结点。父结点包含子结点，子结点又可以是其他结点的父结点。具有相同父结点的子结点是同胞结点。树是结点集合，以层次方式组织树的第一个父结点是根结点(TreeView 控件可以有多个根)。例如，计算机的文件系统可以表示为树。顶级目录(可能是 C:)是根，C:的每个子目录是子结点，每个子文件夹又可以有自己的子结点。

将 TreeView 控件放置到窗体上时是空白的，里面不包含任何结点。打开 Visual Studio 的属性窗口，单击“编辑结点”会弹出如图 11-23 的编辑框。单击“添加根”按钮可以增加一个根结点，单击“添加子级”按钮可以在当前选中的级别下添加一个子结点。结



图 11-23　在 Visual Studio 中编辑 TreeView 结点

点的属性，如 Name 和 Text，可以在窗口右边修改。

添加了数据的 TreeView 结点以层次的方式表示数据间的关系。默认情形下，单击父结点旁边的加号框和减号框可以展开或者折叠一个结点。如果结点不包含子结点，则没有这些框。

图 11-24 显示了图 11-23 中编辑的 TreeView 控件的运行情况，单击加号可以展开。

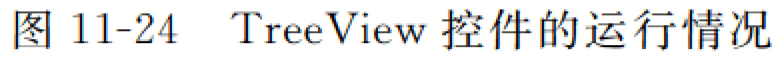


图 11-24　TreeView 控件的运行情况

TreeView 控件中的结点是 TreeNode 类的实例。每个 TreeNode 具有 Nodes 集合（类型 TreeNodeCollection），包含一列其他 TreeNode，是其子结点。Parent 属性返回父结点的引用（根结点返回 null）。

TreeView 的常见属性：

- CheckBoxes：指示是否在树视图控件中的树结点旁显示复选框。true 表示显示；默认值是 false 不显示。
- ImageIndex：树结点显示的默认图像的图像列表索引值。
- ImageList：包含树结点所使用的 Image 对象，是一个包含 Image 对象的集合。
- Indent：每个子树结点级别的缩进距离。
- Nodes：控件的树结点集合。集合中的每一个对象都是 TreeNode 对象。Add 方法可以添加一个结点，Remove 方法可以删除一个结点，Clear 方法可以删除全部结点。注意：删除父结点的同时会删除该父结点下的所有子结点。
- SelectedNode：当前在控件中选定的结点。

TreeNode 的常见属性：

- Checked：指示树结点是否处于选中状态。
- FirstNode：树结点集合中的第一个子树结点。
- ImageIndex：当树结点处于未选定状态时，所显示图像的图像列表索引值。
- Index：树结点在树结点集合中的位置。
- LastNode：最后一个子树结点。
- NextNode：下一个同级树结点。
- Nodes：分配给当前树结点的 TreeNode 对象的集合。
- Parent：当前树结点的父结点。
- PrevNode：上一个同级树结点。
- Text：在树结点标签中显示的文本。

TreeNode 常见的方法：

- Collapse：折叠树结点。
- Expand：展开树结点。
- ExpandAll：展开所有子树结点。
- GetNodeCount：返回子树结点的数目。
- Remove：从控件中移除当前树结点。

在设计模式中，TreeView 的默认事件是 AfterSelect。当所选结点发生变化时，产生该事件。通过程序向 TreeView 控件添加结点时，首先需要创建根结点。使用 Add 方法，可加入一个新的 TreeNode 对象：

```
myTreeView.Nodes.Add(new TreeNode(rootLabel));
```

其中 rootLabel 是 string 类型的，表示在结点上要显示的内容。注意：Add 方法有多个重载，也可以直接给 Add 方法传递一个 string 类型的参数。要将一个结点增加为某个结点的子结点，可在已有的结点中选择一个作为父结点，然后加入：

```
myTreeView.Nodes[index].Nodes.Add(new TreeNode(ChildLabel));
```

在下面的示例中，通过在文本框中输入一个路径(文件夹)，取得该文件夹下所有子文件夹填充的一个 TreeView 控件。除了上述提到的方法和函数外，为了获得文件夹信息，以及处理路径等，使用了在 System.IO 名字空间中的 Directory 类提供的两个静态方法以及 Path 类提供的一个静态方法：

- public static string[] GetDirectories(string path)：参数 path 为返回子目录(文件夹)名称的路径。返回值为在指定路径中的子目录的全名称(包括路径)的数组。
- public static bool Exists(string path)：参数 path 为要测试的路径。如果 path 目录存在，则返回值为 true，否则为 false。
- public static string GetFileNameWithoutExtension(string path)：参数 path 为文件的路径。返回值为字符串，含有文件或文件夹名，但不包括最后的句点“.”以及之后的所有字符。

例 11-10　使用目录信息填充 TreeView 控件。在窗体上放置一个 TreeView 控件、一个文本框和一个 Button 按钮。在文本框中输入一个路径信息，在 TreeView 中将输入的路径作为根结点，填充整个 TreeView 控件。代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace CSHARP11_10
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        ///<summary>
        ///该函数是一个递归函数，取得当前文件夹中的所有
        ///子文件夹，填充 TreeView控件，并且递归该过程
        ///</summary>
```

```
17:         ///<param name="directory">当前的文件夹路径</param>
18:         ///<param name="paprentNode">父结点</param>
19:         private void PopualteTreeView(string directory,TreeNode parentNode)
20:         {
21:             //得到当前文件夹中的所有子文件夹
22:             //并存储在一个 string 数组中
23:             var directoryArray=Directory.GetDirectories(directory);
24:             //填充到 TreeView 控件中
25:             try
26:             {
27:                 if(directoryArray.Length!=0)
28:                 {
29:                     foreach (var dir in directoryArray)
30:                     {
31:                         //得到不包含路径的文件夹的名称
32:                         var subDirectory=
33:                             Path.GetFileNameWithoutExtension(dir);
34:                         TreeNode myNode=new TreeNode(subDirectory);
35:                         parentNode.Nodes.Add(myNode);
36:                         PopualteTreeView(dir,myNode);
37:                     }
38:                 }
39:             }
40:             catch(UnauthorizedAccessException)
41:             {
42:                 parentNode.Nodes.Add("Access denied");
43:             }
44:         }
45:
46:         private void OK_Click(object sender,EventArgs e)
47:         {
48:             //清除 TreeView
49:             dirTreeView.Nodes.Clear();
50:             if(Directory.Exists(dirTextBox.Text))
51:             {
52:                 dirTreeView.Nodes.Add(dirTextBox.Text);
53:                 PopualteTreeView(dirTextBox.Text,dirTreeView.Nodes[0]);
54:             }
55:         }
56:     }
57: }
```

在上述代码中,首先清除了 TreeView 控件(第 49 行)。随后判断文本框中输入的路径是否存在(第 50 行)。如果存在,则作为根结点加入到 TreeView 中(第 52 行),随后调

用函数 PopulateTreeView 将所有的子结点加入。注意，PopulateTreeView 是一个递归函数，对于每一个结点，递归调用该函数，将该结点下的所有子结点加入。程序的运行结果如图 11-25 所示。

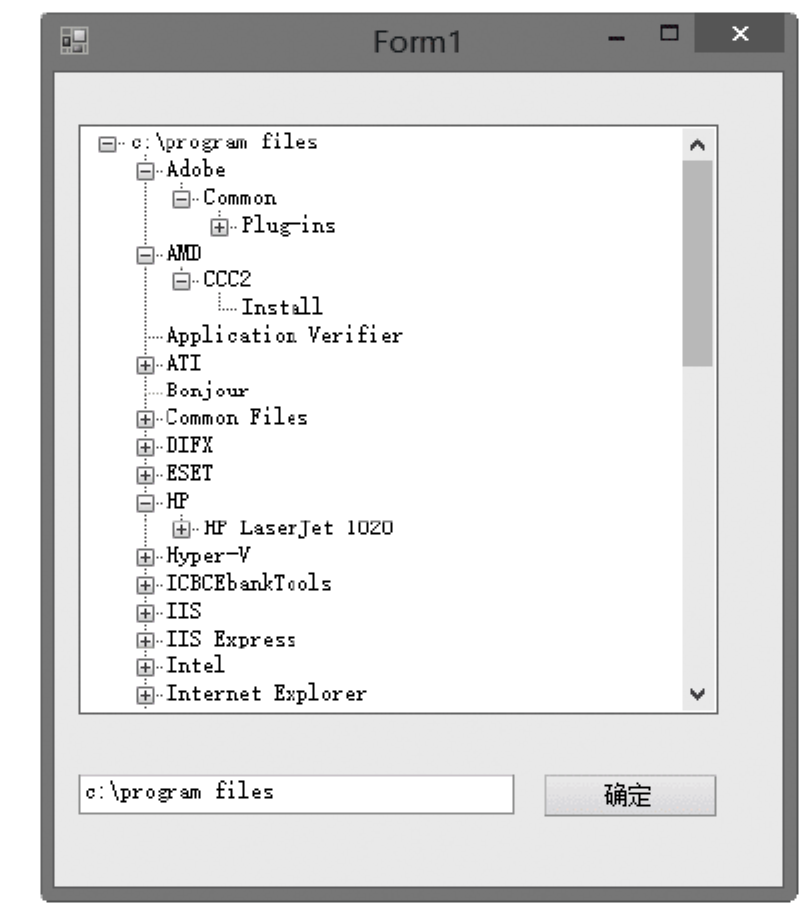


图 11-25 用目录填充 TreeView 的结果

ListView 控件与 ListBox 相似，都显示一个清单，让用户选择一个或几个项目。这两个类的重要差别是 ListView 可以在清单项目旁边显示图标(由 ImageL1st 属性控制)。MultiSelect 属性(布尔值)确定可否选择多个项目。将 CheckBoxes 属性(布尔值)设置为 true，可以包括复选框，使 ListView 的样子类似 CheckListBox。Activation 属性确定用户如何激活项目。这个属性取值为 ItemActivation 枚举，包括 OneClick(单击激活)、TwoClick(双击激活，项目选中时改变颜色)和 Standard(双击激活，项目选中时不改变颜色)。

ListView 可以定义图像。如果 ListView 项目内容要显示图像，则需要 ImageList 组件。ImageLust 组件可以从工具箱中下拉到窗体上，然后在属性窗口选择 Images 属性打开图像集合编辑器(ImageCollectionEditor)。这里可以浏览需要加入 ImageList 的图像，将图像嵌入到应用程序中(成为资源)，发布应用程序时就不需要另外提供图像了。

11.5.4 TabControl 控件

TabControl 显示多个选项卡，这些选项卡类似于档案柜文件夹中的标签。选项卡中可包含图片和其他控件。TabControl 控件可用来产生多页对话框。这种对话框出现在 Windows 操作系统中的许多地方，如显示器控制面板中。

TabControl 控件最重要的属性是 TabPages，它包含单独的选项卡。每个单独的选项卡是一个 TabPage 对象。单击选项卡时，将为相应的 TabPage 对象引发 Click 事件。如果要创建多行选项卡，可将 TabControl 控件的 Multiline 属性设置为 true。如果选项卡尚未以多行方式显示，则设置 TabControl 控件的 Width 属性，使其比所有的选项卡都窄。如果要在控件一侧排列选项卡，将 TabControl 控件的 Alignment 属性设置为 Left 或 Right。一个简单的 TabControl 设计如图 11-26 所示。在设计模式中，给窗体添加了一个 TabControl 控件，随后为这个控件添加了 3 个 TabPage 页面，在每个页面上添加了一组 RadioButton 按钮。这里仅展示了 TabControl 的设计和样式，没有编写代码。在 TabControl 的属性中选择 TabPage 属性，弹出一个 TabPage 的编辑窗口，可以增加或删除一个页，也可以调整页的前后顺序等，如图 11-27 所示。

在添加 RadioButton 按钮时，TabPage 可以作为一个 RadioButton 逻辑组的容器，保证 RadioButton 之间会互相排斥。如果要将多个 RadioButton 组放在一个 TabPage 中，则应该在 TabPage 中用 Panel 或 GroupBox 来分组。

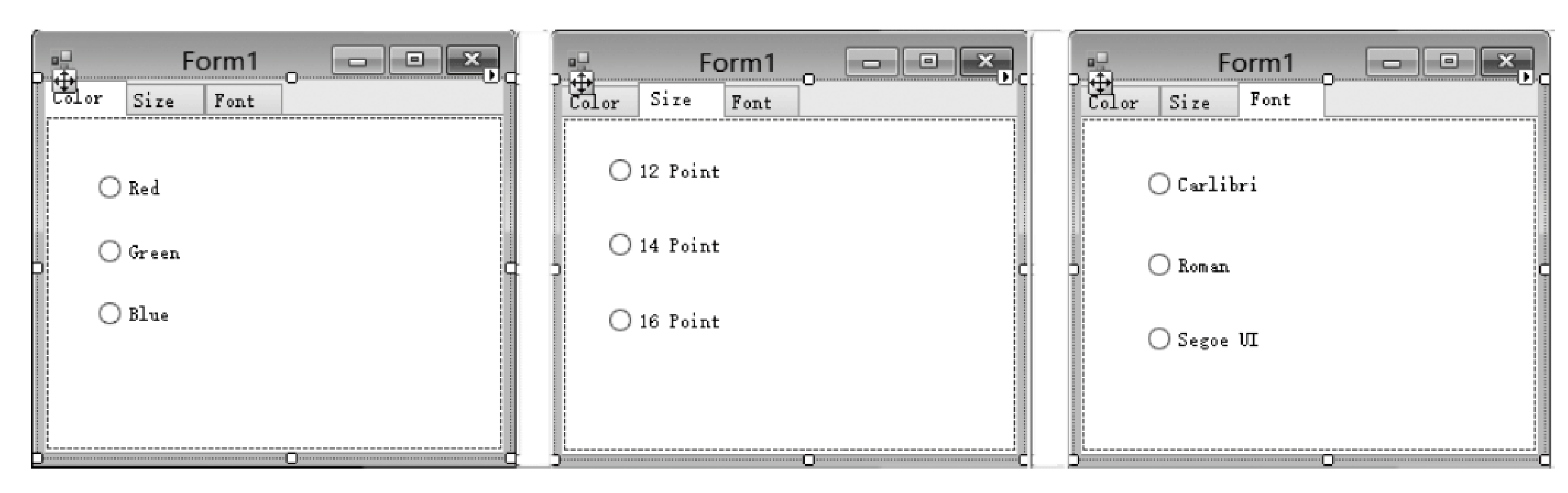


图 11-26　含有 3 个页面的 TabControl 控件



图 11-27　TabPage 集合编辑器

11.6　可视化继承

第 8 章介绍了通过继承其他类创建新类。在本章用继承创建显示 GUI 的窗体，新的 Form 类从 System. Windows. Forms. Form 类派生。这是可视化继承的例子。派生的 Form 类包含 Form 基类的功能，包括基类的属性、方法、变量与控件。派生类还继承所有可视化特性，如缩放、组件布局、GUI 组件间距、颜色与字体等。

可视化继承可以在程序间达到视觉一致性。例如，可以定义一个基类 Form，包含产品徽标、特定背景颜色、预定义菜单栏和其他元素。然后可以在整个程序中使用同一基类 Form，达到统一性和品牌标识。也可以创建控件来继承其他控件。例如，可以创建定制 UserControl（见 11.7 节），从现有控件派生出来新的控件。

在下面的例子中，将定制一个窗口的基类 myBaseForm。该基类是由 Form 类派生而来的。在后续的程序创建中，使用 myBaseForm 类为基类来创建新的窗体程序。要使类能在多个程序中使用，则需要将它放入到类库中，使它可以复用。

例 11-11　创建基类。首先创建一个公用类 myBaseForm，如果类不是公共的，则只能在同一汇编中被其他类使用，也就是说要编译进同一个 dll 文件或者 exe 文件。在本例中，先按原先的方法创建一个正常的 Windows 应用程序，包含一个卷标、一个 PictureBox

(含有一个徽标)以及一个按钮,单击后会显示一些信息。此处将默认的类名 Form1 改为 myBaseForm,如图 11-28 所示。代码如下:

```
using System;
using System.Windows.Forms;

namespace CSHARP11_11
{
    public partial class myBaseForm: Form
    {
        public myBaseForm()
        {
            InitializeComponent();
        }

        private void aboutButton_Click(object sender,EventArgs e)
        {
            MessageBox.Show("CopyRight C#  Programming");
        }
    }
}
```

图 11-28　作为基类的窗口

该程序正确运行后,将其打包为 dll 文件,也就是动态链接库。为此在资源管理器中,在项目名称上右击,选择属性。在弹出的页面左侧选择应用程序,在右侧的输出类型中的下拉清单中选择类库。重新编译项目,这样就可以由原来的 exe 文件生成为 dll 文件。

例 11-12　使用 myBaseForm。新建一个 Windows 应用程序。添加引用,在资源管理器中的项目上右击,在菜单上选择"添加"→"引用"。在弹出的窗口中单击浏览按钮,定位到例 11-11 生成的 dll 文件(通常在项目文件夹的 bin 文件夹中的 debug 文件夹内)。在设计器 Form 窗体上右击,选择查看代码,将生成的 Form1 的基类改为例 11-11 的类:CSHARP11_11.myBaseForm。除非使用 Using CSHARP11_11,否则需要使用类的全名。这时回到窗体设计器,可以看到原来缺省的空白窗体已经被例 11-11 的窗体所取代。在这个窗体上再增加一个 Button 按钮,单击后显示一行文字。注意:设计中,继承的控件的左上角有一个小图标,表示该控件是继承的,无法修改,如图 11-29 所示。

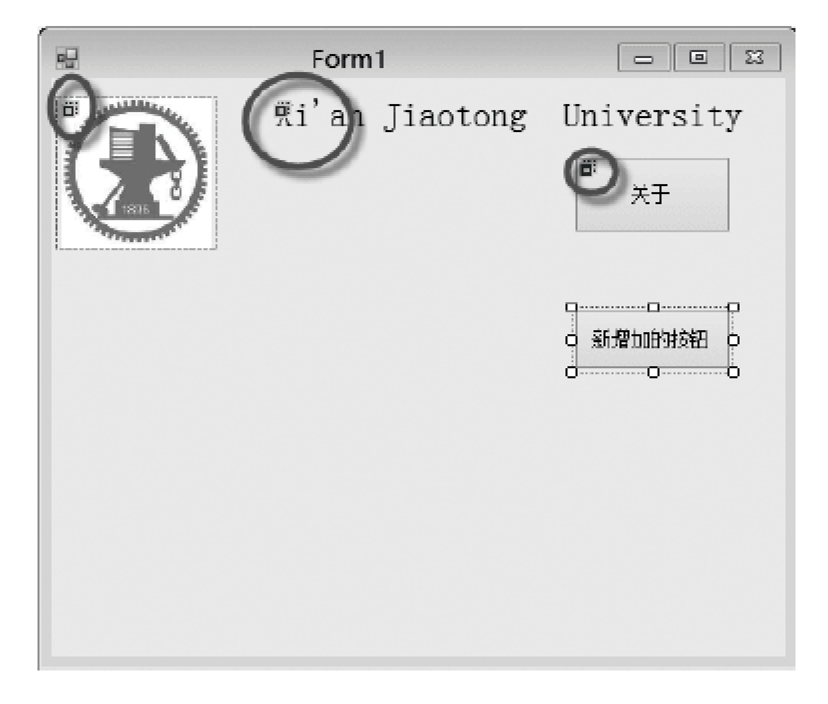


图 11-29　从 myBaseForm 中可视化继承

代码如下:

```
using System;
```

```
using System.Windows.Forms;

namespace CSHARP11_12
{
    public partial class Form1 : CSHARP11_11.myBaseForm
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void newButton_Click(object sender,EventArgs e)
        {
            MessageBox.Show("New Button!");
        }
    }
}
```

11.7 用户定义的控件

.NET 框架允许创建定制控件。这些定制控件放在用户的工具箱中，可以像按钮、卷标和其他预定义控件一样加进窗体面板和组框中。创建定制控件的最简单方法是从现有控件(如卷标)派生，这样可以在现有控件中增加功能，而不必为得到所要功能重新实现现有控件。例如，可以创建新型卷标，它与普通卷标的外观不同。为此可以从 Label 类派生并覆盖 OnPaint 方法。所有控件都包含 OnPaint 方法，系统在重画组件时调用这个方法(如组件缩放时)。OnPaint 方法接受 PantEventArgs 对象，其包含图形信息：属性 Graphics 是要画的图形对象，属性 ClipRectangle 定义控件的矩形边界。系统发出 Paint 事件时，控件基类捕获这个事件，通过多态调用控件的 OnPaint 方法。基类的 OnPaint 实现没有调用，因此要从 OnPaint 实现中显式调用基类的 OnPaint 实现之后再执行定制绘图代码。大多数情况下，这样可以保证先执行原先的绘图代码，再执行定制控件类中定义的代码。如果不让基类 OnPaint 方法执行，则不必调用。

要创建现有控件构成的新控件，可使用 UserControl 类。加进定制控件的控件称为成分控件。例如，编程人员可以创建 UserControl 类，包括按钮、卷标和文本框，各有不同功能(例如，按钮将卷标文本设置成文本框中的文本)。UserControl 类是加进的控件的容器。UserControl 类包含成分控件，因此不确定这些成分控件如何显示。UserControl 的 Onpaint 方法不能覆盖。要控制每个成分控件的样子，就要处理每个控件的 Paint 事件。Paint 事件处理器接受一个 PaintEventArgs 对象，可以在成分控件上画图(直线、矩形等)。

编程人员也可以从 Control 类派生全新的控件。这个类不定义任何特定行为，要用户来定义。Control 类处理与所有控件相关联的项目，如事件和缩放句柄。OnPaint 方法

包含对基类 OnPaint 方法的调用,它调用 Paint 事件处理器,然后要增加代码,在绘制控件时在覆盖 OnPaint 方法中绘制。定制图形有限大的灵活性,但也需要更多的规划。以下是 3 个方法的总结:

- 从 Windows 窗体继承。可以将功能加进现有控件。如果覆盖方法 OnPaint,则需要调用基类的 OnPaint 方法。注意:只能对原控件外观进行补充,不能重新设计。
- 创建一个 UserControl。可以创建 UserControl 类,包括多个现有控件(即组合其功能)。注意:不能覆盖定制控件的 OnPaint 方法,而要把绘图代码放进 Paint 事件处理器中。只能对原控件外观进行补充,不能重新设计。
- 从 Control 类继承。定义全新的控件。覆盖 OnPaint 方法,然后调用基类 OnPaint 方法,包括绘制控件的方法。这个方法可以定制控件外观和功能。

在 Paint 事件中传递 PaintEventArgs 参数,在 PaintEventArgs 对象中含有两个重要的属性。

- Graphics 属性:控件的图形对象(图形对象在第 12 章讲述),用来绘制控件的外观。
- ClipRectangle 属性:指定控件边界的矩形。

有关自定义控件的具体例子请参考其他书籍。

习题

1. 使用文本框、标签和按钮,编写一个计算年利息的程序。用户输入本金、年利率、年限,计算应得的利息。

2. 编写程序在窗体上放置一个 TextBox 控件和 4 个 Button 控件。分别按下这 4 个按钮,可以把文本框的背景色分别设置为红色、蓝色、黑色和绿色。

3. 在窗体上放置 4 个排成矩形的按钮,每个按钮的标题都是“PushMe!”。当用户单击其中一个按钮时,此按钮便会消失,其他 3 个按钮依然存在。

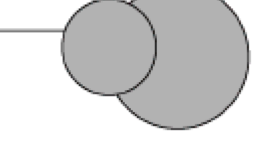
4. 窗体有两个文本框,用户可以在其中输入信息。单击其中一个文本框,它会变成空白,其中的信息转移到另一个文本框中。

5. 在窗体上放置 3 个文本框,按下 Tab 键后,3 个文本框会依次在红、绿、黄 3 种颜色之间循环显示,如同十字路口的红绿灯一样。最初,其中一个文本框是绿色,其他两个是灰色。每当其中一个文本框显示有颜色时(灯亮),其余两个变为灰色(灯灭)。

6. 编写一个程序,含有一个 ComboBox 控件和一个 ListBox 控件。在 ComboBox 中显示 9 种状态名。从 ComboBox 中选择一个项目时,将其从 ComboBox 删除并加 ListBox 中。程序要保证 ComboBox 中至少有一个项目,否则用消息框打印一个消息,然后在用户关闭消息框时终止执行程序。

7. 编写一个程序,让用户在文本框中输入字符串。每个输入的字符串加进 ListBox 中。每个字符串加进 ListBox 时,要保证该字符串已排序(用属性 Sorted)。

第12章



GDI+、菜单、窗体和对话框

12.1 绘图基础知识

12.1.1 坐标系

在 Visual Studio 中，控件放置在窗体对象中，而窗体又放置在屏幕对象中，这些能够放置其他对象的对象称为容器，如窗体、屏幕都是容器。

每个容器都有一个坐标系统，以便为对象的定位提供参考。容器坐标系统的默认设置是容器的左上角为坐标原点，横向向右为 X 轴方向，纵向向下为 Y 轴方向。窗体的原点在紧靠菜单和工具栏（如果有的话）的下方。系统默认的长度单位为像素（Pixel）。

可以使用 Point 来描述 X 和 Y 坐标的有序对，定义二维平面上的一个点。成员 X 代表 X 坐标，成员 Y 代表 Y 坐标。例如：

```
Point pt1=new Point(30,30);
Point pt2=new Point(110,110);
int x=pt1.X;        //值为 30
```

12.1.2 GDI＋绘图

Visual C＃的图形系统 GDI＋（Graphics Device Interface，图形设备接口）是应用程序编程接口，可以理解为用来与特定设备进行交互的一些类。GDI＋可以创建图形、绘制文本以及将图形图像作为对象操作。GDI＋目前是在 Windows 窗体应用程序中以编程方式呈现图形的唯一方法。

所有的 GDI＋相关的类主要分布在 System.Drawing、System.Imaging 和 System.Drawing2D 名字空间中。GDI＋中最主要的对象是 Graphics 对象。它封装了 GDI＋图画板，是 GDI＋绘图中最核心的类。先创建 Graphics 对象，然后才可以使用 GDI＋ 绘制线条和形状，呈现文本或显示与操作图像。Graphics 对象表示 GDI＋绘图表面，是用于创建图形图像的对象。

创建 Graphics 对象有多种方法。最简单的是为窗体的 Paint 事件编写代码。Paint 事件在绘制窗体时发生。可以通过接收 PaintEventArgs 参数中的 Graphics 对象来获得自己的 Graphics 对象：

```
private void Form1_Paint(object sender,PaintEventArgs e)
{
    Graphics g=e.Graphics;
}
```

不仅是窗体，凡是有 Paint 事件的控件（如 ListBox 控件）都可以在该控件的 Paint 事件中取得该控件的 Graphics 对象。Graphics 对象在创建后，可用于绘制线条和形状，呈现文本或显示与操作图像。与 Graphics 对象一起使用的用户对象有：

- Pen：画线、多边形、矩形、弧等外围的轮廓部分。
- Brush：指定颜色、样式、纹理等填充封闭的图形。
- Font：描述字体的样式。
- Color：描述颜色。在 GDI＋中，颜色可以是透明或半透明的。

12.2　在窗体上绘图

在窗体上绘图时，需要设置对象的绘图属性以确定所绘制图形的特征，例如所画线的颜色、宽度、图形的填充样式以及文字的字体等。

12.2.1　画笔和颜色

使用 Pen 对象可以画出不同颜色、样式和宽度的线条。可以在创建一个 Pen 对象时，通过构造函数来设定画线的宽度和颜色，例如：

```
Pen redPen=new Pen(Color.Red);
Pen bluePen=new Pen(Color.Blue,5);
```

第 1 行代码使用红色创建一个新的画笔对象，而第 2 行代码则在创建一个蓝色画笔的同时指定了画笔的宽度是 5 个像素。Pen 类的构造函数和主要属性如下：

构造函数：

- Pen(Brush)：用指定的 Brush（见 12.2.2 节）初始化 Pen 类的新实例。
- Pen(Color)：用指定颜色初始化 Pen 类的新实例。
- Pen(Brush, Single)：使用指定的 Brush 和宽度初始化 Pen 类的新实例。
- Pen(Color, Single)：用指定的颜色和宽度初始化 Pen 类的新实例。

主要属性：

- Brush：用于确定此 Pen 的画刷。
- Color：此 Pen 的颜色。
- DashCap：虚线终点的样式，Pen 绘制的虚线由一系列短划线构成。
- DashOffset：虚线的间隔距离。
- DashStyle：设置 Pen 绘制的虚线的样式。
- EndCap：Pen 绘制的直线终点使用的样式（如箭头、圆点等）。
- StartCap：Pen 绘制的直线起点使用的样式。

- Width：Pen 的宽度。

.NET 框架的 Color 结构用于表示不同的颜色。可以通过 Color 结构访问若干系统预先定义的颜色。例如：

```
Color myFavorColor;          //注意 Color 和 Point 都是值类型的结构

myFavorColor=Color.Red;
myFavorColor=Color.Aquamarine
myFavorColor=Color.LightGoldenrodYellow
```

也可以通过红、绿和蓝三个分量的值(8 位，0～255)来创建一个自己定义的颜色：

```
Color myColor=Color.FromArgb(35, 255, 128);
```

设置好画笔之后，可以使用 Graphics 中的方法 DrawLine 画直线，如下所示：

```
DrawLine(pen,x1,y1,x2,y2);
```

其中 pen 是定义好的画笔。x1、y1 代表直线起始点的 x 坐标和 y 坐标，x2、y2 代表直线结尾点的 x 坐标和 y 坐标。或者：

```
DrawLine(pen, pt1, pt2);
```

pt1、pt2 是 Point 结构类型，分别代表起点和终点的坐标。还可以使用 DrawRectangle 方法绘制矩形：

```
DrawRectangle(pen,x,y,width,height)
```

其中 x、y 代表矩形的左上角的 x、y 坐标值，width 和 height 分别代表这个矩形的宽度和高度。使用 DrawEllipse 函数画圆：

```
DrawEllipse(pen,x,y,width,height);
```

其中 x、y 代表椭圆外接矩形的左上角的 x、y 坐标值，width 和 height 分别代表这个外接矩形的宽度和高度。

例 12-1 画不同的线段。建立工程后，添加窗体的 Paint 消息处理函数，绘制了一些不同的直线。代码如下：

```
using System.Drawing;
using System.Windows.Forms;
using System.Drawing.Drawing2D;

namespace CSHARP12_1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```

```
            InitializeComponent();
        }

        private void Form1_Paint(object sender,PaintEventArgs e)
        {
            Pen myPen=new Pen(Color.CadetBlue,2);
            Graphics g=e.Graphics;
            //画直线
            g.DrawLine(myPen,10,10,200,10);
            //画虚线
            myPen.DashStyle=DashStyle.Dash;
            g.DrawLine(myPen,10,60,200,60);
            //画点划线
            myPen.DashStyle=DashStyle.DashDot;
            g.DrawLine(myPen,10,110,200,110);
            //设置宽度,线尾有箭头,实线
            myPen.DashStyle=DashStyle.Solid;
            myPen.Color=Color.Red;
            myPen.Width=5;
            myPen.EndCap=LineCap.ArrowAnchor;
            g.DrawLine(myPen,10,160,200,160);
            //线头菱形,线尾没有箭头,改变颜色
            myPen.Color=Color.FromArgb(128,0,78);
            myPen.EndCap=LineCap.NoAnchor;
            myPen.StartCap=LineCap.DiamondAnchor;
            g.DrawLine(myPen,10,210,200,210);
        }
    }
}
```

程序运行的结果如图 12-1 所示。

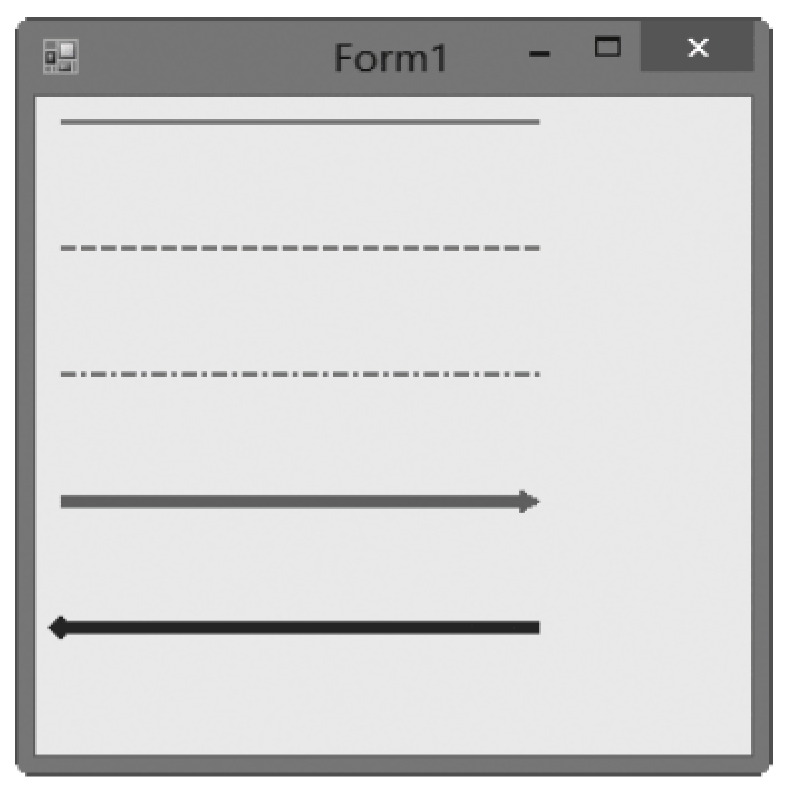


图 12-1　画不同的线段

12.2.2　画刷

封闭图形都包括轮廓线和内部区域。Pen 对象定义轮廓线的属性,而内部区域的属性就由 Brush 对象来定义。GDI+提供了几个笔刷类来填充内部区域,包括 SolidBrush 类、TextureBrush 类和 RectangleGradientBrush 类等。这些类都派生自 Brush 类。

单色刷对应着 SolidBrush 类,用一种颜色填充图形。SolidBrush 类只有一个属性,即 Color 属性。下面的代码声明了一个红色的 SolidBrush 对象:

```
SolidBrush redBrush=new SolidBrush(Color.Red);
```

也可以使用一个图片来填充图形，需要用一个 Bitmap 对象作为构造函数的参数：

```
TextureBrush bitBrush= new TextureBrush(new Bitmap("e:\\MyPhoto.jpg"));
```

其中，MyPhoto.jpg 是存放在 e 盘的一个标准的 jpg 文件。

12.2.3 文字属性

用 Graphics 类的 DrawString 方法可以在图形中引入文字，Font 类定义了文字的特定的格式，如字体、大小、样式属性等。Font 类的构造函数需要三个参数，即字体名、字体大小和字体样式(FontStyle)。例如：

```
Font fontMyWord= new Font("Times New Roman",26,FontStyle.Italic)
```

这条语句声明了一个字体对象 Times New Roman，大小为 26pt(Point，磅)，字体样式为斜体。

12.2.4 绘图

设置好了绘图属性之后，就可以调用 Graphics 中的方法来绘制各种图形了。Graphics 类中提供了大量的方法帮助绘图，主要的绘图方法有：

- Clear：清除整个绘图面并以指定背景色填充。
- DrawArc：绘制一段弧线。
- DrawBezier：绘制贝塞尔样条曲线。
- DrawEllipse：绘制椭圆。
- DrawIcon：绘制指定的 Icon 表示的图像。
- DrawImage：绘制指定的 Image。
- DrawImageUnscaled：使用图像的原始物理大小绘制指定的图像。
- DrawLine：绘制直线。
- DrawPie：绘制扇形。
- DrawPolygon：绘制多边形。
- DrawRectangle：绘制矩形。
- DrawString：绘制文本字符串。
- FillEllipse：填充椭圆的内部。
- FillPie：填充扇形区的内部。
- FillPolygon：填充多边形的内部。
- FillRectangle：填充矩形的内部。

需要注意的是以上的绘图方法基本都有多个不同的重载的版本，如 DrawImage 的重载有 30 个。这样可以根据自己的需要，选择最合适的版本，使用前需要自己参看 MSDN 的帮助文件。Graphics 类中除了绘制方法外，还有对绘制区域裁剪、复制和变换的方法。

例 12-2 绘图的演示。在这个例子中，添加 form 的 Paint 事件处理程序，在 Paint 事件处理程序中，示例了一些绘图方法，以及画刷、字体、位图的绘制。

程序代码如下：

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace CSHARP12_2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            //需要重绘整个窗口
            this.ResizeRedraw=true;
        }

        private void Form1_Paint(object sender,PaintEventArgs e)
        {
            Graphics g=e.Graphics;
            //刷子
            SolidBrush blackBrush=new SolidBrush(Color.Black);
            SolidBrush whiteBrush=new SolidBrush(Color.White);

            //需要显示的图像
            Bitmap photoBitmap=new Bitmap(".\\MyPhoto.jpg");
            //需要使用的字体
            Font myFavorFont=new Font("New Times Roman",13,
                FontStyle.Italic);
            //窗体当前的大小
            int width=this.ClientSize.Width;
            int height=this.ClientSize.Height;
            //将窗体填充为黑色
            g.FillRectangle(blackBrush,0,0,width,height);

            int imageWidth=photoBitmap.Width;
            int imageHeight=photoBitmap.Height;
            //缩放图像,按比例显示在窗口的中间
            //图片显示在窗口中的左上角坐标
            double ratio=Math.Min((double)this.Height/imageHeight,
                (double)this.Width/imageWidth);
            int x=(int)(this.Width - imageWidth * ratio)/2;
            int y=(int)(this.Height - imageHeight * ratio)/2;
            Rectangle formRect=new Rectangle(x,y,(int)(imageWidth * ratio),
                (int)(imageHeight * ratio));
            Rectangle imageRect=new Rectangle(0,0,imageWidth,imageHeight);
```

```
44:                //显示图片
45:                g.DrawImage(photoBitmap,formRect,imageRect,
46:                        System.Drawing.GraphicsUnit.Pixel);
47:                //输出文字
48:                g.DrawString("Photo By Cui Shuning",
49:                    myFavorFont,whiteBrush,x+20,y+formRect.Height-20);
50:
51:            }
52:        }
53: }
```

程序运行结果如图12-2所示。



图12-2 在窗体上绘图的示例

在代码中，使用填充矩形方法FillRectangle(31行)用黑色刷子填充了背景。DrawImage方法(45行)将图片显示到窗口。在DrawImage方法中使用了2个矩形区域，表示要显示图片的哪一部分，在此是整个图片(43行)以及按比例显示到窗口中的矩形区域(41行)。使用DrawString方法输出一个字符串(48行)：

```
DrawString(String,Font,Brush,x,y);
```

5个参数分别是要输出的字符串、字体、颜色刷、字符串的起始位置坐标。

12.3 在控件上绘图

不但可以在窗体上绘图，而且也可以在大部分控件上绘图。在控件上绘图之前，同样需要创建一个Graphics对象。此时Graphics对象通常都是通过可在上面绘图的控件的CreateGraphics方法来创建的。

例12-3 在控件上绘图。在窗体上放置一个Button控件和一个TextBox控件，TextBox控件的MutiLines属性设置为true。为Button的Click事件编写代码，为窗体的OnPaint事件编写代码。

程序代码如下：

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace CSHARP12_3
{
    public partial class Form1 : Form
    {
        public Form1()
```

```
10:            {
11:                InitializeComponent();
12:            }
13:            bool drawing=false;
14:            private void drawButton_Click(object sender,EventArgs e)
15:            {
16:                drawing=true;
17:                Invalidate(new Rectangle(textBox1.Location,textBox1.Size));
18:            }
19:
20:            private void Form1_Paint(object sender,PaintEventArgs e)
21:            {
22:               if(drawing)
23:               {
24:                   Graphics g;
25:                   SolidBrush brushRed=new SolidBrush(Color.Red);
26:                   SolidBrush brushGreen=new SolidBrush(Color.Green);
27:                   SolidBrush brushYellow=new SolidBrush(Color.Yellow);
28:
29:                   g=textBox1.CreateGraphics();
30:                   g.FillEllipse(brushRed,42,2,28,28);
31:                   g.FillEllipse(brushGreen,12,2,28,28);
32:                   g.FillEllipse(brushYellow,72,2,28,28);
33:               }
34:            }
35:        }
36: }
```

在上述程序中，下述语句：

```
g=textBox1.CreateGraphics();
```

通过调用 TextBox 控件的成员函数 CreateGraphics()得到了一个 Graphics 对象，随后利用该对象在 TextBox 控件内画了 3 个圆。注意，此时的绘图函数的坐标原点在 TextBox 控件的左上角。程序运行结果如图 12-3 所示。

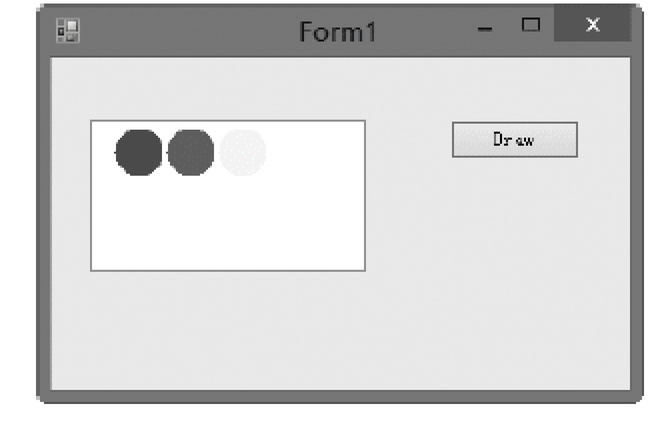


图 12-3　在控件上绘图

需要注意的是不要把绘图代码放在按钮单击的事件处理中，原因是：虽然这样单击按钮可以得到与图 12-3 一样的效果，但是当程序被最小化后恢复时，就没有代码来绘制这 3 个圆了，控件上是空白的。所以把绘制代码放入到了窗体的 OnPaint 中，使得窗口在改变时可以再次绘制。按钮单击的代码中，使用布尔变量 drawing 控制是否绘制（13 行），而 14 行的 Invalidate 则是通知窗体有所改变，需要重绘，也就是激发了一个 OnPaint 消息，使得 Form1_Paint 方法能够立即被执行。Invalidate 中的矩形参数通知窗体需要重绘的区域，不含参数的 Invalidate 方法则重绘整个窗口。较

小的、更精确的重绘区域可以在重绘时减少窗体的闪烁。

12.4 菜单

一般来讲，菜单有两种形式：一种是通常位于应用程序顶部的，它提供了对应用程序所有功能的访问途径；另一种是使用鼠标在某个对象上右击时，此时弹出的菜单称为上下文相关菜单（ContextMenu）。

12.4.1 菜单的基本概念

所谓主菜单，就是可供选择的命令项目列表，它位于菜单栏上，例如 Visual Studio 集成开发环境界面中提供了“文件”、“编辑”、“视图”、“窗口”、“帮助”等多个菜单。单击某个菜单选项立刻会下拉出该菜单选项的列表。每个菜单标题的后面会提供一个带下划线的字符，按下 Alt 键不放并键入带下划线的字符 也可拉出该菜单项的列表，例如按 Aft+F 组合键和单击“文件”菜单标题的效果是一样的，都可以打开“文件”菜单。

在打开的下拉菜单中，选择一个带有省略号“…”的命令时，将打开一个对话框，在对话框中选择所需要的信息后单击“确定”按钮，即可执行该菜单命令。

在菜单项中，分隔线的作用就是将菜单项分组，使作用相近的菜单项放置在一起。例如，在“编辑”菜单中，常将“剪切”、“复制”和“粘贴”命令用分隔线分为一组。建议在自己设计菜单时最好也要遵守这个规则，这样用户在使用应用程序的菜单命令时会感到非常方便，

某些菜单项的右边有一个向右的小箭头 这是带有子菜单项的标志，当选取这样的菜单项时，将打开下一级子菜单，可以从子菜单中选择要执行的命令。在 Visual C＃ 中，最多可以使用 6 级子菜单。

还有一些菜单项不执行命令，而是代表一种状态。如在 Word 2003 中，“显示段落标记”的菜单项前带有“√” 标志，表示显示内容的状态，即将文档的段落标记显示出来。在程序运行中，菜单有隐藏、无效和正常三种状态。隐藏菜单是在窗口运行时不出现在菜单栏上的菜单；无效菜单是指下拉菜单中以灰色显示的菜单项，它表示该菜单项在目前这种状态下不能执行；正常状态是指可以正常使用的菜单。

12.4.2 设计并使用菜单

设计菜单和设计窗体按钮的过程一样，首先在界面上放置好菜单，然后设置菜单的属性，最后根据需要编写菜单命令所触发的事件的响应代码。

例 12-4 简单的文本编辑器。从工具箱的菜单及工具栏选项卡中将 MenuStrip 控件拖放到窗体上，该控件会自动安置到窗体的顶端，同时在设计窗口的底部显示控件名，如图 12-4 所示。在该应用程序中创建菜单。菜单栏包含了 3 个菜单：File、Edit 和 View，以及一些菜单项和子菜单项。单击窗体顶端标为“请在此处输入”的框，选中它，然后再单击，使其处于编辑状态，输入“文件(&F)”，这就创建了文件菜单。&F 表示在字符 F 下加下划线并将 F 作为文件菜单的快捷字符，这样就可以用 Alt 键和带下划线的字母

来访问这个菜单。在刚才输入的菜单的下方有一个框，在框内输入“新建”，这样就建立了“新建”菜单项。在输入框中输入减号(—)，可以产生一个菜单项分隔符。设计好的文件菜单如图 12-5 所示。随后分别选择菜单的各项，打开属性编辑器，将它们的 Name 属性分别设置为 fileMenu、fileNewMenu、fileOpenMenu 和 fileExitMenu。以同样的方法设计编辑菜单，编辑菜单中共有 5 项，分别为撤消、复制、剪切、粘贴和全选。编辑菜单的 Name 属性为 editMenu，5 个菜单项的 Name 属性分别为 editUndoMenu、editCopyMenu、editCutMenu、editPasteMenu 和 editSelectAllMenu。为了给这 5 个菜单项建立快捷键，分别将它们的 Shortcut 属性设定为 CtrlZ、CtrlC、CtrlX、CtrlV 和 CtrlA。

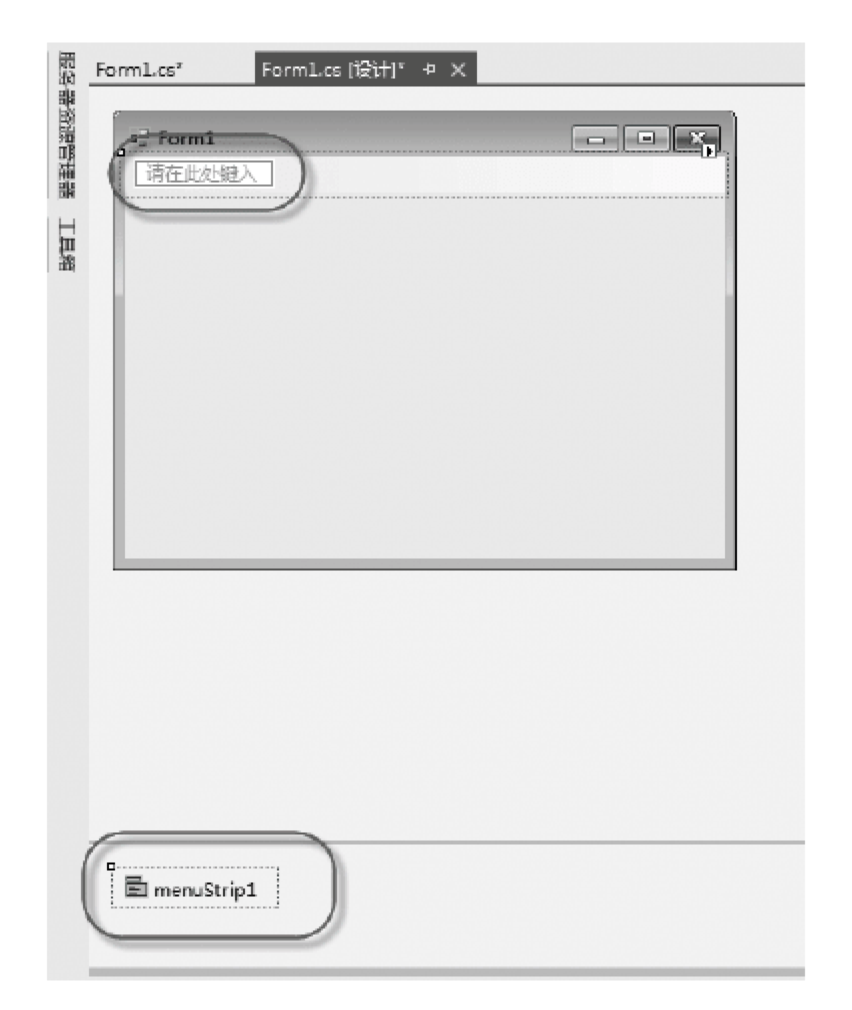


图 12-4　创建菜单



图 12-5　设计好的文件菜单

建立最后一个视图菜单。视图菜单下只有一个菜单项是“工具栏”。视图菜单的 Name 属性为 viewMenu。菜单项工具栏的 Name 属性为 viewToolMenu。在工具栏菜单项下添加 3 个子菜单项，分别为格式、地址和链接。Neme 属性分别为 viewToolFormatMenu、viewToolAddressMenu 和 viewToolLinkMenu。这 3 项是让用户选择是否显示这些工具栏，因此将它们的 Checked 属性设置为 True，即给菜单项加复选标记，表明该菜单项启动时是有效的。在菜单下放置一个 TextBox 控件，设置其 Name 属性的值为 txtEdit，Anchor 属性的值为 Top、Bottom、Left、Right，Multiline 属性的值为 True，Scrollbar 属性的值为 Both。

将 TextBox 的大小调整为充满这个窗体。一个新的属性是 Anchor(锚定)，几乎所有的控件都有这个属性。它的作用是在程序运行时，保证 Anchor 所指定的边和窗体的相对位置不变。由于指定了所有 4 条边，因此用户改变窗口的大小时，TextBox 控件也会随之改变(也可以使用 Dock 属性设置为 Fill)。

设计完成后，分别给各菜单项添加相应的事件处理代码。整个程序的代码如下：

```
using System;
using System.Windows.Forms;

```

```
namespace CSHARP12_4
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void fileNewMenu_Click(object sender,EventArgs e)
        {
            txtEdit.Text= "";
            txtEdit.Focus();
        }

        private void fileOpenMenu_Click(object sender,EventArgs e)
        {
            //此处的内容在第 13 章
        }

        private void fileExitMenu_Click(object sender,EventArgs e)
        {
            Application.Exit();
        }

        private void editUndoMenu_Click(object sender,EventArgs e)
        {
            txtEdit.Undo();
        }

        private void editCopyMenu_Click(object sender,EventArgs e)
        {
            txtEdit.Copy();
        }

        private void editCutMenu_Click(object sender,EventArgs e)
        {
            txtEdit.Cut();
        }

        private void editPasteMenu_Click(object sender,EventArgs e)
        {
            txtEdit.Paste();
```

```
47:            }
48:
49:            private void editSeletedAllMenu_Click(object sender,EventArgs e)
50:            {
51:                txtEdit.SelectAll();
52:            }
53:
54:            private void viewToolBarFormatMenu_Click(object sender,EventArgs e)
55:            {
56:                viewToolBarFormatMenu.Checked= !viewToolBarFormatMenu.Checked;
57:            }
58:
59:            private void viewToolBarAddressMenu_Click(object sender,EventArgs e)
60:            {
61:                viewToolBarAddressMenu.Checked= !viewToolBarAddressMenu.Checked;
62:            }
63:
64:            private void viewToolBarLinkMenu_Click(object sender,EventArgs e)
65:            {
66:                viewToolBarLinkMenu.Checked= !viewToolBarLinkMenu.Checked;
67:            }
68:        }
69: }
```

在文件菜单中的“新建”菜单项添加一个单击事件过程，选择“新建”菜单后，程序将文本框清空，并通过调用文本框的 Focus 方法给文本框设置焦点（第 16 行）。Edit 菜单中的第一个菜单项是“撤销”菜单项。在代码中，直接调用了 TextBox 控件的 Undo 方法而完成了撤销功能（第 31 行）。类似地可以使用 TextBox 的 Copy 方法、Cut 方法、Paste 方法等。视图菜单下的工具栏菜单项下有 3 个子菜单项：格式、地址和链接。启动程序时，默认它们有一个复选标记。单击这个菜单项时应该在有标记和无标记之间切换。在单击该子菜单项时，可使 MenuViewToolFormat 子菜单项的 Checked 属性在 true 与 false 间转换，即该子菜单项名旁的复选标记在被选中和不被选中之间转换（第 66 行）。

12.4.3　设计上下文菜单

上下文菜单（ContextMenu）是单击鼠标右键时弹出的菜单。它的设计方法与主菜单中的子菜单的设计方法相同。Visual Studio 2013 提供了一个标准控件 ContextMenuStrip，来帮助编程人员使用 ContextMenu。ContextMenu 控件可以与窗体上的其他控件关联，也可以与窗体本身关联。将上下文菜单与窗体或控件关联的方法是使用窗体或控件的 ContextMenuStrip 属性。工具箱中的大多数控件都有 ContextMenuStrip 属性，把窗体或控件的 ContextMenuStrip 属性设置为前面定义的 ContextMenuStrip 菜单控件名即可。

可以为例 12-4 添加一个 ContextMenu，当鼠标在 TextBox 上右击时出现。在视图设计器中，从工具箱中找到 ContextMenuStrip 控件，并将其放置到窗体上，将其命名为 editContextMenu。与 MenuStrip 控件一样，它将会被添加到开发环境的底部。

选中 ContextMenuStrip 控件，会在窗体顶部出现上下文菜单的编辑器。和编辑 MenuStrip 一样，在上下文菜单中输入 5 项，分别为撤销、复制、剪切、粘贴和全选。这 5 项的 Name 属性分别为 undoCMenu、copyCMenu、cutCMenu、pasteCMenu 和 selectAllCMenu。同时将它们的 Shortcut 属性分别设定为 CtrlZ、CtrlC、CtrlX、CtrlV 和 CtrlA。选中 txtEdit 控件，将它的 ContextMenuStrip 属性设置为 editContextMenu。这样 txtEdit 控件就和刚才的菜单关联起来了，在该控件上单击右键会弹出这个上下文菜单。要使弹出的菜单真正有用，还需要为每一个菜单项的 Click 事件编写代码。编写代码的方法和 MainMenu 是一样的，在此就不再重复了。

在某些情况下，应使相应的菜单有效或无效，例如在没有选定任何文本的情况下，剪切和复制菜单项是无效(灰色)的。可以为菜单的 DropDownOpening 事件编写程序，该事件在菜单弹出前发生。在这个事件代码中，检查 TxtEdit 的属性 SelectionLength。若该属性为 0，则表示当前没有文本被选定，因此将复制和剪切的 Enbled 属性设为 false 以使它们弹出时是灰色的。代码如下：

```
private void viewMenu_DropDownOpening(object sender,EventArgs e)
{
    if (txtEdit.SelectionLength==0)
    {
        editCopyMenu.Enabled=false;
        editCutMenu.Enabled=false;
    }
    else
    {
        editCopyMenu.Enabled=true;
        editCopyMenu.Enabled=true;
    }
}
```

12.5 窗体

窗体是程序的一个重要对象，也是 Visual C# 可视化程序设计的基础。对于用户来说，窗体的界面就是应用程序，程序的可用性完全依赖于窗体界面。掌握主窗体常用项目的设置是非常重要的，有利于快速按照要求构建应用程序框架。

窗体分为单文档窗体和多文档窗体两种类型。单文档窗体是指每次只能在窗体中打开一个画面，如 Windows 提供的记事本程序就是单文档窗体。多文档窗体是指一次能打开多个画面的窗体，如 Microsoft Word 即属于典型的多文档窗体应用程序。

12.5.1　在项目中加入新的窗体

在前面介绍的程序中，通常只需要设计一个窗体就行了，但大多数程序不只需要一个窗体，而是要给程序加入多个窗体。Visual Studio 为应用程序的第一个窗体起的默认名是 Form1，其后默认名为 Form2、Form3，依次类推。在设计 Windows 窗体时，应该给默认的窗体起一个合适的名字，这既有利于代码的编写和调试，也有利于程序的维护。

窗体的 Text 属性决定了窗体标题栏显示的内容，应该将其改为更有意义的字符。此外，在解决方案资源管理器中，可以看到对应于某个窗体的文件名通常和该窗体的名字是一致的。同时也是和该窗体对应的类名是一致的。最好不要直接修改类名 Form1。建议使用以下步骤进行修改：

(1) 打开类视图窗口，选中 Form1 类，然后再在属性窗口更改类名。这样做的好处是 Visual Studio 会自动更改和 Form1 相关的资源文件的设置。

(2) 在解决方案资源管理器中选中 Form1.cs，在属性窗口更改文件名。

要在项目中加入第二个窗体，可在项目菜单中选择添加新建项，将弹出添加新项的对话框。在其中选择 Windows 窗体，并将名称改为 MySecond 后单击“确定”按钮。可以看到一个新的名为 MySecond 的窗体加入到了项目中。

12.5.2　窗体的显示和隐藏

窗体提供了多个方法，可以通过这些方法对窗体进行加载、显示、隐藏、卸载等操作。Show 方法用来显示一个已经装入内存的窗体。如果调用 Show 方法时指定的窗体没有加载，将自动加载该窗体。例如，要显示窗体 Form2，可以使用下面的语句：

```
Form2.Show()
```

Hide 方法用来隐藏显示在屏幕上的窗体。隐藏窗体时，将从屏幕上删除窗体，并将其 Visible 属性设置为 False。用户将无法访问隐藏窗体上的控件，但是运行中的应用程序仍然可以使用隐藏窗体的控件。如果调用 Hide 方法时窗体还没有加载，那么 Hide 方法将加载该窗体，但不显示它。

12.5.3　标准对话框

在应用程序开发过程中，对话框的设计往往是一个非常重要的内容，应用程序的许多交互和设置功能都是由它来实现的。例如，打开或保存文件、要求用户输入适当的数据和设置应用程序内容等情况出现时，都需要利用对话框来实现。.NET 提供了一些内置的对话框，这些对话框提供了大多数 Windows 应用程序中常见的标准对话框，可以利用这些对话框来完成一些常用的任务。

Visual Studio 在其工具箱中提供了多个对话框控件，大大方便了开发人员在自己的应用程序中使用打开文件、保存文件、打印设置、颜色和字体设置等常用的标准对话框。这些标准的对话框自动生成一些常用的设置项，而且能够按照常用的方式响应用户的

操作。

在文件操作过程中，经常要打开或保存文件，所以大部分应用程序都需要设置一种方式来打开或保存文件。在 Visual C# 中提供了 OpenFiledialog 和 SaveFilelog 两个控件，通过它们可以在文件的打开和保存操作中为用户选择文件提供方便。通过设置 OpenFileDialog 控件的属性来决定对话框的标题、默认路径和文件类型等内容。最主要的 Filter 属性用来对文件进行过滤。例如，要在对话框中只显示后缀名为 SYS 的文件，可以设定 Filter 的值为 SYS(＊.SYS)|＊.SYS，其中“|”号前的字符串将作为文件类型列表中的文字说明部分，“|”号后的字符串指定显示在文件列表中的文件类型。如果限定的类型为两种以上，各个类型还应以成对的方式建立。例如，要限定打开的类型为＊.TXT、＊.EXE 和＊.SYS，在 Filter 属性栏中应输入如下：

```
Text(* .Text)| * .TXT | EXE(* .EXE) | * .EXE | SYS(* .SYS) | * .SYS
```

用户选择了某个文件并单击“确定”按钮后，对话框会关闭，选择的文件名(带有完整的路径)将保存在 OpenFileDialog 的 FileName 属性中。仅此而已，文件不会被真正打开。要显示一个 OpenFileDialog 对话框可以使用它的 ShowDialog 方法。

SaveFileDialog 控件的使用和设置与 OpenFileDialog 控件基本相同，只是多了一些在保存文件时的属性。SaveFileDialog 控件不同于 OpenFileDialog 控件的属性主要有两个，其中 CreatePrompt 用于处理不存在的新文件。如果该属性的值为 True，则在用户指定的文件不存在时询问用户是否建立新文件，默认该值为 False，即不询问用户。

OverwritePrompt 用于处理已经存在的文件。如果该属性的值为 True，则在用户指定的文件已经存在时询问用户是否覆盖文件；如果该值为 False，则不询问用户。SaveFileDialog 控件的方法与 OpenFileDialog 控件完全相同，可按照处理 OpenFileDialog 控件的方法来处理 SaveFileDialog 控件。例如要显示一个 SaveFileDialog 控件，可使用 SaveFileDialog 控件的 ShowDialog 方法。

ColorDialog 控件是用于设置颜色的对话框。在一些具有图像处理和文本编辑功能的应用程序中，常常需要添加颜色设置对话框，以便于为图像或文本设置颜色。利用 ColorDialog 控件创建的颜色对话框是一个标准的 Windows 颜色设置对话框，它支持几百万种颜色。

FontDialog 控件用于选择字体。字体设置是大部分应用程序，特别是具有文本编辑能力的应用程序的必备功能。应用程序中，字体设置是通过 FontDialog 控件调用一个名称为字体的对话框来实现的。利用这个对话框，可以完成所有关于字体的设置。

12.5.4 消息对话框

消息对话框，简称消息框(MessageBox)，用来在应用程序运行时将提示、警告和错误等信息及时通知用户，并等待用户回应。例如，修改了数据而没有保存就试图关闭应用程序时，系统就会显示带有信息、警告图标和按钮的消息框，提示将丢失未保存的数据。用户可单击某个按钮(如 OK 或 Cancel)来继续或取消这步操作。表 12-1 列出了一些可以在消息框中显示的图标。

表 12-1　消息对话框中的图标

图　　标	常　　数	图　　标	常　　数
	IconAsterisk		IconExclamation
	IconInformation		IconWarning
	IconHand		IconQuestion
	IconStop		

表 12-2 列出了一些可以在消息框中显示的按钮。

表 12-2　消息对话框中的按钮

常　　数	说　　明
AbortRetryIgnore	显示“终止”、“重试”和“忽略”按钮的消息框
OK	显示“确定”按钮的消息框
RetryCancel	显示“重试”和“取消”按钮的消息框
YesNo	显示“是”和“否”按钮的消息框
YesNoCancel	显示“是”、“否”和“取消”按钮的消息框

可以通过调用 Show 方法显示消息框：

```
DialogResult Show(string text,string caption,MessageBoxButtons buttons,MessageBoxIcon icon)
```

其中 text 代表所要显示的信息，该参数是必需的。caption 代表显示在消息框的标题栏中的字符串，该参数为可选的。buttons 参数可用来指定显示在消息框中的可用按钮。icon 参数可用来指定显示在消息框中的可用图标。

例 12-5　下面将通过一个简单的例子来说明这些对话框的使用。编写一个简单的徒手绘图程序。该程序有 3 个顶层菜单、文件、颜色和线形。在文件菜单下，共有 3 个项目：

(1) 新建菜单(Name 属性 fileNewMenu)：将清除当前的图像，新建一个指定宽度和高度的图像，背景色为白色。

(2) 保存菜单(Name 属性 fileSaveMenu)：将目前窗体中的图像保存为 JPG 文件。

(3) 退出菜单(Name 属性 fileExitMenu)：退出程序。

在颜色菜单下只有一项，选择颜色(Name 属性 colorSelectedMenu)；线形菜单下有 2 项互斥的选择，选择虚线或者实线。

在程序中，当按下鼠标不放时，将沿着鼠标的轨迹，用指定的颜色和线形画线。默认的线形是实线，颜色是黑色。需要为每个菜单项编写消息处理方法，以及鼠标的消息处理和窗体的 OnPaint 消息。

需要注意的是，程序并不是在窗体上绘图，保存窗体上的内容需要和 WindowsAPI 交互，这已经超出了本书的范围。本程序是在一个 Bitmap 对象上工作，然后将该对象显示在窗口中。Bitmap 对象可以新建，也可以是打开的一个图像文件。程序代码如下：

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Drawing.Drawing2D;

namespace CSHARP12_5
{
    public partial class Form1 : Form
    {
        private string saveFileName;
        private Color color= Color.Black;                //默认黑色
        private DashStyle dashStyle= DashStyle.Solid;    //默认实线
        private Bitmap bitmap;                            //打开的位图,如果没有值为 NULL
        private Point prePoint;                           //鼠标移动的前一个点
        private Point nextPoint;                          //鼠标移动的下一个点
        private bool mouseCapture= false;                 //鼠标捕获标记

        public Form1()
        {
            InitializeComponent();
        }

        private void fileNewMenu_Click(object sender,EventArgs e)
        {
            //弹出新建对话框,让用户输入高度和宽度
            NewFile newFileDialog= new NewFile();
            if(newFileDialog.ShowDialog()= = DialogResult.OK)
            {
                bitmap= new Bitmap(newFileDialog.ImageWidth,
                    newFileDialog.ImageHeight);
                Graphics g= Graphics.FromImage(bitmap);
                //设置底色为白色
                Brush whiteBrush= new SolidBrush(Color.White);
                g.FillRectangle(whiteBrush,0,0,newFileDialog.ImageWidth,
                    newFileDialog.ImageHeight);
                Invalidate();
            }
        }

        private void fileSaveMenu_Click(object sender,EventArgs e)
        {
            SaveFileDialog saveFileDialog= new SaveFileDialog();
            saveFileDialog.Filter= "jpg(* .jpg)| * .jpg";
            if(saveFileDialog.ShowDialog()= = DialogResult.OK)
```

```
            {
                saveFileName=saveFileDialog.FileName;
                //当前窗口的图像保存为 jpg 文件
                bitmap.Save(saveFileName);
            }

        }

        private void Form1_Paint(object sender,PaintEventArgs e)
        {
            if(bitmap!=null)
            {
                e.Graphics.DrawImage(bitmap,0,0);
            }
        }

        private void fileExitMenu_Click(object sender,EventArgs e)
        {
            Application.Exit();
        }

        private void colorSelectedMenu_Click(object sender,EventArgs e)
        {
            ColorDialog colorDialog=new ColorDialog();
            if(colorDialog.ShowDialog()==DialogResult.OK)
            {
                color=colorDialog.Color;
            }
        }

        private void lineSolidMenu_Click(object sender,EventArgs e)
        {
            lineSolidMenu.Checked=true;
            dashStyle=DashStyle.Solid;
            lineDashMenu.Checked=false;
        }

        private void lineDashMenu_Click(object sender,EventArgs e)
        {
            lineDashMenu.Checked=true;
            dashStyle=DashStyle.Dash;
            lineSolidMenu.Checked=false;
        }
        //鼠标事件的处理
```

```
        private void Form1_MouseDown(object sender,MouseEventArgs e)
        {
            //判断是否左键,并且在图像范围内按下
            if (e.Button==MouseButtons.Left && e.X<bitmap.Width
                && e.Y<bitmap.Height)
            {
                mouseCapture=true;
                prePoint=new Point(e.X,e.Y);
            }
        }

        private void Form1_MouseMove(object sender,MouseEventArgs e)
        {
            if (mouseCapture)
            {
                if (e.X<bitmap.Width && e.Y<bitmap.Height)
                {
                    Graphics g=Graphics.FromImage(bitmap);
                    nextPoint=new Point(e.X,e.Y);
                    Pen pen=new Pen(color,1);
                    pen.DashStyle=dashStyle;
                    g.DrawLine(pen,prePoint,nextPoint);
                    //应该构造要刷新的矩形区域,减小闪烁,本程序未做
                    Invalidate();
                }
                prePoint=nextPoint;
            }
        }

        private void Form1_MouseUp(object sender,MouseEventArgs e)
        {
            mouseCapture=false;
        }
    }
}
```

程序运行时,单击"新建"按钮将弹出一个新的窗体,如图 12-6 所示。



图 12-6　输入高度和宽度

从 Visual Studio 的项目菜单中选择添加 Windows 窗体,将窗体命名为 NewFile。在该窗体上有两个 TextBox(Name 属性分别为 widthTextBox 和 heightTextBox)和一个取消按钮以及一个确定按钮(Name 属性分别为 cancelButton 和 OKButton)。设置两个按钮的 DialogResult 属性分别为 Cancel 和 OK。这样当这个窗体使用 ShowDialog 方法显示时,单击不同的按钮会返回不同的值(程序

27～30 行）。在 NewFile 这个窗体中设置了两个 public 的属性，当单击“确定”按钮后将文本框输入的值存储在这两个属性中。要注意的是外部无法直接访问这两个文本框，因为其是 private 的（代码第 29、34 行）。NewFile 窗体的代码如下：

```
01: using System;
02: using System.Windows.Forms;
03:
04: namespace CSHARP12_5
05: {
06:     public partial class NewFile : Form
07:     {
08:
09:         public int ImageWidth { get;set;}
10:         public int ImageHeight { get; set; }
11:         public NewFile()
12:         {
13:             InitializeComponent();
14:         }
15:
16:         private void OKButton_Click(object sender,EventArgs e)
17:         {
18:             ImageWidth=Convert.ToInt32(widthTextBox.Text);
19:             ImageHeight=Convert.ToInt32(heightTextBox.Text);
20:         }
21:     }
22: }
```

新建一个文件后，程序的运行界面如图 12-7 所示。

在颜色菜单下单击选择颜色，弹出颜色选择对话框。选择一个颜色后保存在字段 color 中（代码第 66～73 行）。注意：我们是在一个 Bitmap 对象上绘图，而不是在窗体上，所以绘图的 Graphics 对象是从 Bitmap 对象创建的（代码第 110 行）。要保存图像，只需要调用 Bitmap 自己的 Save 方法即可（代码第 48 行）。在代码中可以看到，所有的对话框都是调用自身的 ShowDialog 方法来显示自己，确认用户单击“确定”按钮之后，将用户的选择保存在某个属性或者字段中。

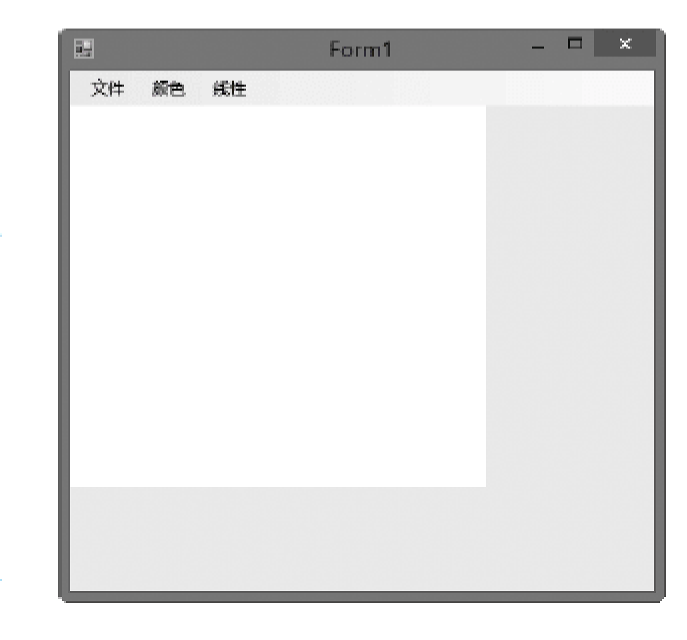


图 12-7　程序的运行界面

12.5.5　多文档程序

一个 MDI 程序至少需要用到两个窗体。其中一个窗体作为容器，另一个窗体作为模板。一般将作为容器的窗体称为父窗体。在 MDI 程序中可以同时打开多个文档，每个文档用一个单独的窗体显示。每一个这样单独的窗体称为一个子窗体。一般而言，所有的子窗体都是类似的，使用窗体模板来创建。从类的角度讲，所有的子窗体是一个类的不同

实例。下面通过一个例子来具体讲述 MDI 程序的创建。

例 12-6 创建一个 MDI 程序，它能够同时打开多幅图片。为程序添加一个菜单，菜单的顶层有两项，文件和窗口。在文件菜单下有两项，分别是打开和退出。在窗口的菜单下有 3 项，分别是水平平铺、垂直平铺和层叠。重新合理地命名这些菜单项的 Name 属性。

选中当前的窗体，它将成为父窗体或是容纳子窗体的容器。欲使它成为容器，需要将窗体的 IsMdiContainer 属性设置为 True。此时，窗体的背景色变为深灰色。从项目菜单中选择添加新项，出现添加新项的对话框。在模板中选择 Windows 窗体，在名称一栏中填入一个合适的名字如 ChildWindows 后单击“打开”按钮，新的窗体将被加入到项目中。将新窗体的 AutoScroll 属性改为 True。将一个 PictureBox 控件放置到新窗体上，并将它的 SizeMode 属性设置为 AutoSize。该选择使得 PictureBox 控件自动将大小调整为和图片实际尺寸相同。将 Dock 属性设置为 Fill，充满窗口。

为文件菜单下的打开项的 Click 事件编码，它简单地显示一个打开文件对话框。我们已经知道，用户的选择将存放在该控件的 FileName 属性中。在用户确定选择了一个图片文件后将对此作出响应。显示图片的第一步是生成一个新的子窗口，通过产生一个新的子窗体类 Child 的实例 formChildWindows，得到一个新的窗体。接下来需要指出该子窗体的父窗体是哪一个，在本例中，它是当前的窗体。因此，将 formChildWindows 的 MdiParent 属性设置为 this。随后在 PictureBox 控件中打开文件，并调用窗体的 Show 方法显示新的窗体。程序代码如下：

```
using System;
using System.Windows.Forms;

namespace CSHARP12_6
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void fileOpenMenu_Click(object sender,EventArgs e)
        {
            OpenFileDialog openFileDlg=new OpenFileDialog();
            if(openFileDlg.ShowDialog()==DialogResult.OK)
            {
                ChildWindows formChildWindows=new ChildWindows();
                formChildWindows.MdiParent=this;
                formChildWindows.LoadPicture(openFileDlg.FileName);
                formChildWindows.Show();
            }
        }
```

```
24:
25:             private void fileExitMenu_Click(object sender,EventArgs e)
26:             {
27:                 Application.Exit();
28:             }
29:
30:             private void viewTileHorizontallyMenu_Click(object sender,
                        EventArgs e)
31:             {
32:                 LayoutMdi(MdiLayout.TileHorizontal);
33:             }
34:
35:             private void viewTileVerticallyMenu_Click(object sender,EventArgs e)
36:             {
37:                 LayoutMdi(MdiLayout.TileVertical);
38:             }
39:
40:             private void viewStackMenu_Click(object sender,EventArgs e)
41:             {
42:                 LayoutMdi(MdiLayout.Cascade);
43:             }
44:         }
45: }
```

将父窗体的 IsMdiContainer 属性设置为 True 后，整个窗体会成为暗灰色，如图 12-8 所示。

同时可以通过调用父窗体的 LayoutMdi 方法对多个打开的子窗口进行排列（代码第 30～43 行），层叠窗口的效果如图 12-9 所示。

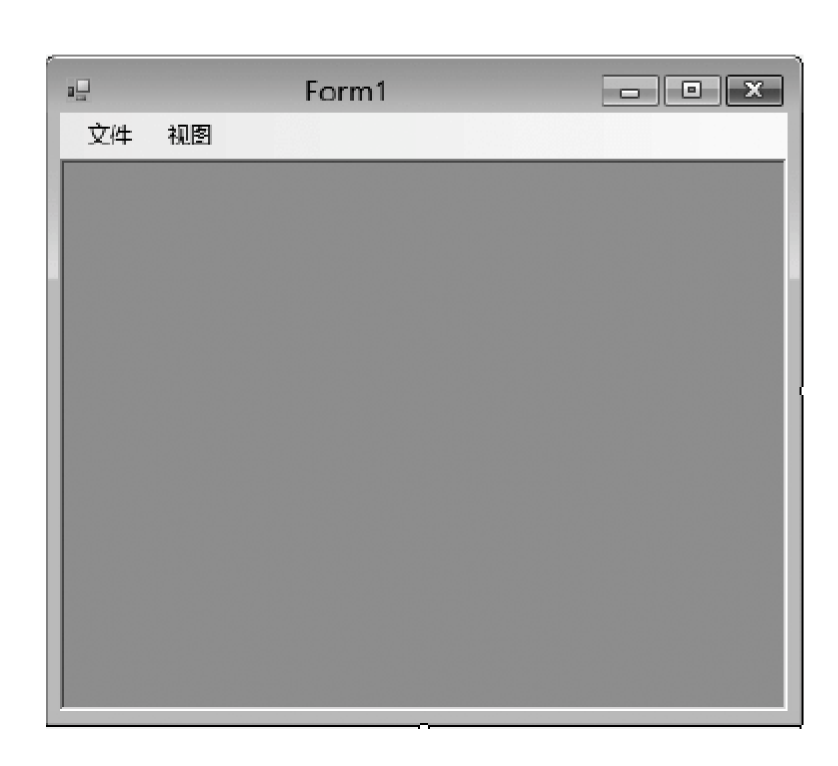


图 12-8　程序的界面设计



图 12-9　打开 3 个窗口并且层叠的效果

在 ChildWindows 中需要添加一个公开的方法 LoadPicture，添加这个方法的主要原因是 PictureBox 在 ChildWindows 中是私有变量，无法在外部访问，通过添加该 Public 型的函数来调用。ChildWindows 类的代码如下：

```
using System.Drawing;
using System.Windows.Forms;

namespace CSHARP12_6
{
    public partial class ChildWindows : Form
    {
        public ChildWindows()
        {
            InitializeComponent();
        }
        public void LoadPicture(string fileName)
        {
            pictureBox.Image= Image.FromFile(fileName);
        }
    }
}
```

习题

1. 使用图形方法，在 Form 上画出 5 条不同颜色的直线并形成一个多边形。

2. 使用图形方法，在 Form 上画一条经过(200,200)、(256,87)、(87,9)、(22,108)这四个点的折线。

3. 使用图形方法，在 Form 上画一个椭圆，并用纹理刷填充。

4. 将前面 3 题组合在一起，并设计一个菜单来完成各项功能。

5. 在第 4 题的程序中再增加一个“帮助”菜单，里面有“关于”一项。单击该项，显示一个关于版权的对话框。

6. 重新编写第 2 题的程序。使用颜色对话框让用户可以选择颜色。

7. 改写例 12-4，使它可以同时编辑多个文件。

第13章 文件和流

13.1 文件和流的基本概念

13.1.1 文件的概念

文件是指记录在外部介质上的数据的集合。Visual Studio 中与文件操作有关的类在命名空间 System.IO 中，因此在程序的开头需要引入这个命名空间。一般而言，文件可分为两类：

(1) 文本文件：文本文件是完全由字符和与字符的显示格式有关的控制符（如空格符、回车符及换行符等）构成。文本文件很容易在不同的操作系统、不同的应用软件之间通用，只要这些系统和软件遵循相同的字符代码标准即可。文本文件常用 TXT 作为其扩展名，常见的文本文件扩展名还有 BAT、HTM 等。

(2) 二进制文件：二进制文件为非文本文件，这类文件一般依赖于特定的软件，有各种不同的内部格式。扩展名为 COM、EXE 的程序文件、存放图像数据的 BMP 文件、存放声音数据的 WAV 文件等都是二进制文件。

13.1.2 流的概念

文件一般存放在磁盘上，将磁盘上的文件内容载入内存，称为读文件。相反，将数据保存到磁盘上称为写文件。读写文件是通过文件流的方法来实现的。流是一个动态的概念，是指数据从出发地"流"到目的地。抽象地说，流就是一个字节序列，其最重要的特点就是对于流的操作是按照流中字节的先后顺序来进行的。事实上，流有很多具体的形式：文件和操作文件的程序之间的字节序列是一个流，网络通信中传递的数据也是一个流。

流可以分成输入和输出两类。比如从键盘输入到内存的是输入流，从内存到打印机的是输出流。例如，在读写文本文件时，相应的输入流就是把文件中的文本传递到文本编辑器的流，实现这个流的类是 StreamReader。输出流就是把文本编辑器中的文本传递到文本文件的流，实现这个流的类是 StreamWriter。

13.2 文本文件的读写

在.NET中,与文件操作有关的类在命名空间System.IO中,System.IO命名空间中定义了许多对象用于支持数据流的处理。要使用System.IO命名空间,应在窗体代码的顶部加入语句

```
using System.IO;
```

用于文本文件读写的主要有两个对象:StreamWriter对象和StreamReader对象。

13.2.1 StreamWriter

使用StreamWriter对象,可以将数据写入文本文件,如果所指定的文件不存在,则可以创建一个新文件。要创建一个StreamWriter对象,可使用new语句:

```
StreamWriter swFile=new StreamWriter("C:\\MyFile.txt");
StreamWriter swFile=new StreamWriter("C:\\MyFile.txt",True);
```

有多种方法可以创建一个StreamWriter实例。在上面的第一个语句中,打开文件MyFile.txt准备向该文件写入。此时,StreamWrite的构造函数接收要写入文件的路径作为唯一的一个参数,该参数是String类型的。在第二个语句中除了路径参数外,还有一个Boolean型的变量作为第二个参数。在文件已经存在的情况下,该参数如果为True,则新写入的数据被追加到文件尾;否则新数据将覆盖旧数据。如果文件不存在,则创建新文件。StreamWriter对象中,核心的方法有:

- Write方法:调用StreamWriter对象的Write方法,可以往文件中写入一个字符串。
- WriteLine方法:调用StreamWriter的WriteLine方法,可以往文件中写入一个字符串和一个换行符(写入一行)。
- Close方法:释放StreamWriter对象,并关闭打开的文件。

下面的代码创建一个StreanWriter类,并把一些文本添加到新创建的文本中(写入的文本是唐代诗人李白的《古意》):

```
StreamWriter swFile=new StreamWriter("C:\\MyFile.txt");
swFile.WriteLine("君为女萝草,妾作兔丝花。");
swFile.WriteLine("轻条不自引,为逐春风斜。");
swFile.WriteLine("百丈托远松,缠绵成一家。");
swFile.WriteLine("谁言会面易,各在青山崖。");
swFile.Close();
```

13.2.2 StreamReader

使用StreamReader读取标准文本文件的各行信息。创建一个StreamReader对象时,可以指定一个带有路径的文件名。一旦对象创建成功,便可以从该文本文件中读取字

符了：

```
StreamReader srFile=new StreamReader ("C:\\MyFile.txt");
```

StreamReader 对象常用的方法有：

- Read 方法：从文件(流)中读入下一个字符。
- ReadLine 方法：从文件(流)中读入下一行字符。
- Close 方法：关闭打开的文件(流)。
- ReadToEnd 方法：从文件(流)的当前位置读到文件(流)的结尾。
- Peek 方法：返回文件(流)中的下一个字符,但并不读入该字符。

采用上面的方法,可以将上一节创建的文件读入到一个字符串中

```
StreamReader srFile=new StreamReader ("C:\\MyFile.txt");
String strIn;
While(srFile.Peek() >-1)
{
    strIn+=srFile.ReadLine();
}
```

例 13-1　读写文本文件。将 TextBox 的内容写入文件或从指定的文件中读出内容显示在 TextBox 中。在窗体上放置两个 Button 控件和一个 TextBox 控件。为两个 Button 控件的 Click 事件编写代码。程序代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace CSHARP13_1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void writeButton_Click(object sender,EventArgs e)
        {
            SaveFileDialog saveFileDlg=new SaveFileDialog();
            saveFileDlg.Filter="txt(*.txt)|*.txt";
            if(saveFileDlg.ShowDialog()==DialogResult.OK)
            {
                try
                {
                    StreamWriter swFile=new StreamWriter
```

```
                                   (saveFileDlg.FileName);
                    swFile.Write(fileTextBox.Text);
                    swFile.Close();
                }
                catch
                {
                    MessageBox.Show("Write ERROR!!");
                }
            }
        }

        private void readButton_Click(object sender,EventArgs e)
        {
            OpenFileDialog openFileDlg=new OpenFileDialog();
            openFileDlg.Filter="txt(*.txt)|*.txt";
            if(openFileDlg.ShowDialog()==DialogResult.OK)
            {
                try
                {
                    StreamReader srFile=new StreamReader
                        (openFileDlg.FileName);
                    fileTextBox.Clear();
                    fileTextBox.Text=srFile.ReadToEnd();
                    srFile.Close();
                }
                catch
                {
                    MessageBox.Show("Read ERROR!!");
                }
            }
        }
    }
}
```

需要特别注意的是：在写入和读出操作完成之后，一定要调用 Close 方法关闭文件，只有这样才能确保写入和读出文件的正确完成。写入和读出的代码放到了 try 的结构中，捕获写入、读出中可能出现的异常。

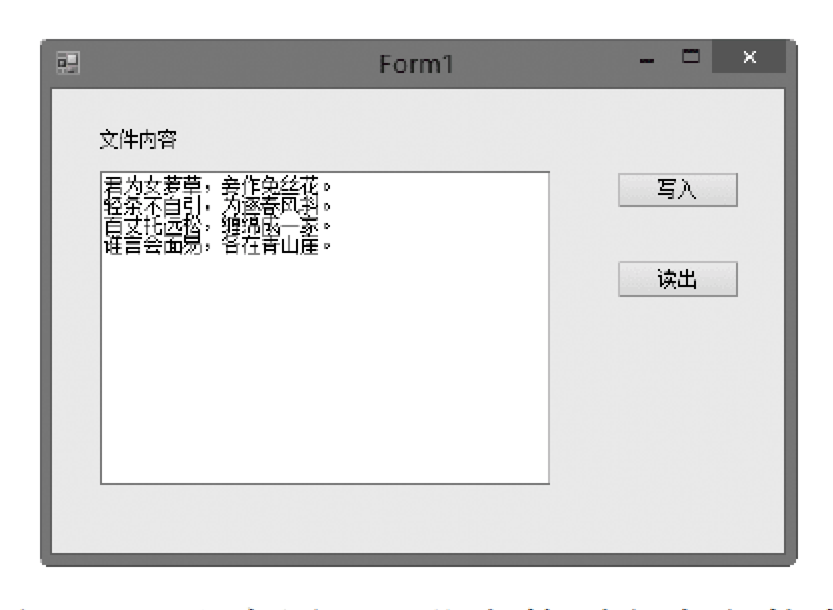


图 13-1　程序写入一些字符到文本文件中

首先在文本框中输入一些字符，单击写入按钮将打开标准的保存文件对话框，选择保存位置后，将文本框中的内容写入文件，如图 13-1 所示。

在资源管理器中找到写入的文件，直接用

记事本打开，验证写入过程是正确的。随后直接用记事本创建一个新的文件，保存为 MyFile2.txt。运行程序打开该文件，验证正确性。

13.3　二进制文件的读写

上一节介绍了文本文件的读写，这种文件可以通过“记事本”程序打开来查看内容。除此以外，还有一种文件形式，即二进制文件，本节介绍如何读写二进制文件。

13.3.1　FileStream

可以使用 FileStream 对象打开（或创建）文件。它的构造函数常用的形式如下：

```
public FileStream(String path, FileMode mode, FileAccess access)
```

其中：

- path 用于指定要打开（或创建）的文件的路径及文件名。
- mode 用于指定文件打开的模式，可以是 Create（创建）、Open（打开）等。
- access 用于指定文件访问的目的，可以是 Read（读）、Write（写）或 ReadWrite（读和写）。

例如：

```
FileStream fsRW=new FileStream("E:\\MyFile.bin",FileMode.Open,FileAccess.Read);
```

其中 FileMode 和 FileAccess 是枚举变量。上述语句表示打开 E:\MyFile.bin 读出其中的内容。可以使用 FileStream 中的 Read 和 Write 方法来对文件进行读写。Read 和 Write 方法的调用方法相似，如下所示：

```
override int Read(unsigned char[] array, int offset, int count);
```

Read 和 Write 方法具有一样的 3 个参数，分别是：

- array Byte 型数组，存放准备写入文件流的数据（Write 中），或是从文件流中读出的数据（Read 中）。
- offset 准备从数组中向文件写入或读出的数据在数组中的偏移量。
- count 写入或读出的最大字节数。

此外，Write 方法是 void 型的，没有返回值。而 Read 方法则有一个返回值，表示读入缓冲区中的总字节数。如果当前的字节数没有所请求的那么多，则总字节数可能小于所请求的字节数；或者如果已到达流的末尾，则为零。

FileStream 中还可以使用 ReadByte 方法和 WriteByte 方法一次向文件流中读出或写入一个字节：

```
override void WriteByte(Byte value);
override int ReadByte();
```

ReadByte 将读入的一个字节转换为整数返回。此外，Close 方法用于关闭打开的文

件流，而 Seek 方法则用于在文件流中定位：

```
override long Seek(long offset, SeekOrigin origin);
```

调用该方法，可以将文件的当前位置移动到指定位置。其中，

- origin 指定指针移动的方向，可有三种选择。
- Begin 从文件头往后开始移动指针。
- Current 从指针的当前位置往后开始移动指针。
- End 从文件尾往前开始移动指针。
- offset 指定相对于 origin 的移动偏移量。

例如：

```
fsRW.Seek(20,Seekorigin.Begin)
```

是指将当前位置设置为从文件头开始的第 20 个字节处。

13.3.2 BinaryWriter 和 BinaryReader

FileStream 对象只提供了字节方式的写入。在构造了 FileStream 对象后可以将该对象进一步构造为 BinaryWriter 和 BinaryReader 对象，以获取更高级的功能。需要从一个存在的流来构造 BinaryWriter 和 BinaryReader 对象，例如：

```
FileStream fsRW=new FileStream("E:\\MyFile.bin",FileMode.Open,FileAccess.Read);
BinaryWriter bwMyFile=new(fsRW);
```

BinaryWriter 提供了很多重载的 Write 方法来方便对文件的写入，部分如下：

- void Write(Boolean)：将 1 字节 Boolean 值写入当前流。
- void Write(Byte)：将一个无符号字节写入当前流。
- void Write(Char())：将字符数组写入当前流。
- void Write(Decimal)：将一个十进制数值写入当前流。
- void Write(Double)：将 8 字节浮点值写入当前流。
- void Write(Short)：将 2 字节有符号整数写入当前流。
- void Write(Integer)：将 4 字节有符号整数写入当前流。
- void Write(Long)：将 8 字节有符号整数写入当前流。

在 BinaryReader 中，则有各自的读入方法，例如：

- ReadBoolean：从当前流中读取 Boolean。
- ReadByte：从当前流中读取下一个字节。
- ReadBytes：从当前流中将 count 个字节读入字节数组。
- ReadChar：从当前流中读取下一个字符。
- ReadDecimal：从当前流中读取十进制数值。
- ReadDouble：从当前流中读取 8 字节浮点值。
- ReadSingle：从当前流中读取 4 字节浮点值。
- ReadString：从当前流中读取一个字符串。字符串有长度前缀，一次 7 位地被编

码为整数。

同时要注意的是：这些方法从当前流读入数据或写入数据后，均会根据读入或写入数据所占的字节数修正流的当前位置。

例 13-2 将职工的工资信息写入一个二进制文件，并读出来。设计如图 13-2 所示的窗体。



图 13-2 设计好的窗体

窗体左边共使用了 4 个 TextBox 控件来分别输入工号、姓名、年龄和工资，并在其旁边分别用 Label 标签做了说明。写入按钮每次将用户填入的数据写入一个二进制文件中，数据总是追加在文件的结尾处。

窗体右边使用了一个 TextBox 控件，用来显示读出结果。单击写入按钮之后，程序将左边 4 个文本框中的内容写入到文件中，通常称这样一组信息为一个记录。若每条记录有固定的长度（占的字节数相等），则可以使在文件中随机查询和定位一条记录变得更加简单。

因此，设定了 6 个长度的序号字段、4 个长度的名字，并且对用户的输入进行检查。序号不足 6 位将会要求重新输入，超过 6 位的将被截去。名字不足 4 个字的将被补上空格。另外，在获得输入前，首先去掉了输入前后的空格。这是通过 String 的成员函数 Trim 来完成的。程序代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace CSHARP13_2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void writeButton_Click(object sender,EventArgs e)
        {
            char[] workID=new char[6];
            var strTemp=workIDTextBox.Text.Trim();
            if (strTemp.Length<6)
            {
                MessageBox.Show("序号必须为 6位!请重新输入。
                超过 6位的部分将被截去!");
```

```
            return;
        }
        else
        {
            strTemp.CopyTo(0,workID,0,6);
        }
        char[] name=new char[4];
        strTemp=nameTextBox.Text.Trim();
        if (strTemp.Length==0)
        {
            MessageBox.Show("请输入姓名!");
            return;
        }
        else
        {
            if (strTemp.Length <=4)
            {
                strTemp.CopyTo(0,name,0,strTemp.Length);
            }
            else
            {
                strTemp.CopyTo(0,name,0,4);
            }
        }

        if (ageTextBox.Text.Trim().Length==0 ||
        salaryTextBox.Text.Trim().Length==0)
        {
            MessageBox.Show("请输入年龄或工资!");
            return;
        }

        int age=Convert.ToInt32(ageTextBox.Text);
        float salary=Convert.ToSingle(salaryTextBox.Text);
        //写入文件中
        FileStream fsSalary=new FileStream(".\\Salary.bin",
            FileMode.Append,FileAccess.Write);
        BinaryWriter bwSalary=new BinaryWriter(fsSalary);
        //写入数据
        bwSalary.Write(workID);
        bwSalary.Write(name);
        bwSalary.Write(age);
        bwSalary.Write(salary);
        //关闭文件
```

```
            bwSalary.Close();
        }

        private void readButton_Click(object sender,EventArgs e)
        {
            readTextBox.Clear();
            FileStream fsSalary;
            try
            {
                fsSalary=new FileStream(".\\salary.bin",
                    FileMode.Open,FileAccess.Read);
            }
            catch (FileNotFoundException e1)
            {
                MessageBox.Show(e1.ToString());
                return;
            }
            //读文件
            BinaryReader brSalary=new BinaryReader(fsSalary);
            char[] workID=new char[6];
            char[] name=new char[4];
            int age;
            float salary;
            //string strTemp;

            while (brSalary.PeekChar() >-1)
            {
                workID=brSalary.ReadChars(6);
                name=brSalary.ReadChars(4);
                age=brSalary.ReadInt32();
                salary=brSalary.ReadSingle();

                readTextBox.Text+="工号：";
                for (int i=0; i<6; i++)
                    readTextBox.Text+=workID[i];
                readTextBox.Text+="\r\n"+"姓名：";
                for (int i=0; i<4; i++)
                    readTextBox.Text+=name[i];
                readTextBox.Text+="\r\n"+"年龄："+age.ToString();
                readTextBox.Text+="\r\n"+"工资："+salary.ToString();
                readTextBox.Text+="\r\n";

            }
            brSalary.Close();
```

```
            }
        }
}
```

程序对所有的输入检查完后，分别将其复制到本地的局部变量中。随后构造了一个FileStream流，再由这个流构造了BinaryWriter对象。接着调用BinaryWriter对象的Write方法将数据写入到文件中。第一次将创建新文件，以后将数据追加在文件尾。

读出则把所有人员信息显示在右侧的TextBox中。程序从第一条记录开始，每次读出一条记录。在构造FileStream流的时候使用Try语句，以捕获当要打开的文件不存在的时候产生的异常。事实上在进行任何文件操作的时候，均应该放入Try语句中。

13.3.3 序列化

上一节介绍了如何将数据写入文件。数据输出到磁盘文件时，会失去某些信息，如每个值的类型。例如，如果从文件中读取数值“3”，则不能判断这个值来自int、字符串或decimal。磁盘中只有数据，没有类型信息。如果读取这个数据的程序“知道”对应数据的对象类型，则可以直接将数据读取为这个类型的对象。例如，在例13-2中，我们知道一条记录的数据类型组成以及先后次序，因此，可以正确读写数据。有时更简单的方法就是读取和写入整个对象。C#提供了这种机制，称为对象序列化。序列化对象表示为字节序列，包括对象数据和关于对象类型及对象中所存放数据类型的信息。序列化对象写入文件之后，可以从文件读取并去序列化，即用对象类型及对象中所存放数据类型的信息在内存中重建对象。

BinaryFormatter类（名字空间SystemRuntime.Serialization.FormattersBinary）可以读取和写入流中的整个对象。BinaryFormatter方法Serialize将对象表示写入文件。BinaryFormatter方法Deserialize读取这个表示并重建对象。这两个方法在序列化或去序列化遇到错误时都抛出SerializationException异常。这两个方法都要求用Stream对象作为参数，使BinaryFormatter能够访问正确的流。

例13-3 使用序列化的方法重写例13-2。代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;


namespace CSHARP13_3
{
    [Serializable]
    public class WorkRecord
    {
        public char[] workID=new char[6];
        public char[] name=new char[4];
```

```
        public int age;
        public float salary;
    }
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void writeButton_Click(object sender,EventArgs e)
        {
            //使用对象来存储每一条的记录
            WorkRecord workRec= new WorkRecord();
            var strTemp= workIDTextBox.Text.Trim();
            if (strTemp.Length< 6)
            {
                MessageBox.Show("序号必须为 6位!请重新输入。
                超过 6位的部分将被截去!");
                return;
            }
            else
            {
                strTemp.CopyTo(0,workRec.workID,0,6);
            }
            strTemp= nameTextBox.Text.Trim();
            if (strTemp.Length= = 0)
            {
                MessageBox.Show("请输入姓名!");
                return;
            }
            else
            {
                if (strTemp.Length <= 4)
                {
                    strTemp.CopyTo(0,workRec.name,0,strTemp.Length);
                }
                else
                {
                    strTemp.CopyTo(0,workRec.name,0,4);
                }
            }

            if (ageTextBox.Text.Trim().Length= = 0 ||
```

```
                salaryTextBox.Text.Trim().Length==0)
            {
                MessageBox.Show("请输入年龄或工资!");
                return;
            }

            workRec.age=Convert.ToInt32(ageTextBox.Text);
            workRec.salary=Convert.ToSingle(salaryTextBox.Text);
            //写入文件中
            FileStream fsSalary=new FileStream(".\\Salary.ser",
            FileMode.Append,FileAccess.Write);
            //序列化
            BinaryFormatter binFomatter=new BinaryFormatter();
            binFomatter.Serialize(fsSalary,workRec);
            //关闭文件
            fsSalary.Close();
        }

        private void readButton_Click(object sender,EventArgs e)
        {
            readTextBox.Clear();
            FileStream fsSalary;
            try
            {
                fsSalary=new FileStream(".\\salary.ser",
                FileMode.Open,FileAccess.Read);
            }
            catch (FileNotFoundException e1)
            {
                MessageBox.Show(e1.ToString());
                return;
            }
            //读文件并且去序列化
            BinaryFormatter readFomatter=new BinaryFormatter();
            int count=0;
            WorkRecord workRec=
                (WorkRecord)readFomatter.Deserialize(fsSalary);
            readTextBox.Text+="工号:";
            for (int i=0; i<6; i++)
                readTextBox.Text+=workRec.workID[i];
            readTextBox.Text+="\r\n"+"姓名:";
            for (int i=0; i<4; i++)
                readTextBox.Text+=workRec.name[i];
            readTextBox.Text+="\r\n"+"年龄:"+workRec.age.ToString();
```

```
102:                readTextBox.Text+="\r\n"+"工资："+workRec.salary.ToString();
103:                readTextBox.Text+="\r\n";
104:                count++;
105:                fsSalary.Close();
106:            }
107:        }
108: }
```

在代码中，为了方便使用 BinaryFormatter 类，引入了名字相应的名字空间(代码第 4 行)；同时将要存储的数据组织在了一个单独的对象中，并申明该对象是可序列化的(代码第 9 行)。注意：该例与例 13-2 不同，在文件中只写入了一个对象的数值，也只读出了一个对象的数值，原因是如果存储多个对象的话，则需要自己计算文件中对象的个数，FileStream 类不好判断是否到达了流的末尾。

13.4　目录和文件操作

.NET 还提供了大量的方法对磁盘上的文件或目录进行操作和管理。如创建文件夹，移动、删除和复制文件等。这些方法所属的类有些是静态类，因此无须生成一个类的实例便可以使用这些方法。

13.4.1　目录操作

Directory 类提供了操作一个目录所需要的大部分方法。Directory 类中的方法全部是静态的，因此无须生成 Directory 的实例便可以使用这种方法。以下是一些常用的方法：

- CreateDirectory(string Path)：该方法按照 Path 所指定的路径创建一个新的目录。如果 Path 指定的路径格式不对或者不存在均会引发异常。最好将该方法置于 try 语句中。
- Delete(string Path, bool recursive)：该方法删除 Path 所指定的目录。如果 recursive 为 false，则仅当目录为空时删除该目录。若为 true，则删除目录下的所有子目录和文件。
- Exists(string Path)：该方法测试 Path 所指定的目录是否存在。若存在，则返回 true，否则返回 false。
- GetDirectories(string Path)：该方法得到 Path 所指定的目录中包含的所有子目录。结果以字符串数组的形式返回，数组中的每一项对应一个子目录。
- GetFiles(string Path)：该方法和 GetDirectories 方法类似，返回目录中所有的文件名。
- Move(string sourceDirName, string destDirName)：该方法将一个目录中的内容移动到一个新的位置。sourceDirName 指定要移动的目录，destDirName 指定移动到何处。

• GetLogicalDrives()：返回计算机中的所有逻辑驱动器名称。

例 13-4 读出磁盘上的目录和文件。程序启动后，将在 Combox 控件中显示所有的驱动器列表。选择了一个驱动器后，将在左边目录的 ListBox 控件中显示该驱动器下的所有文件夹，在右边的 ListBox 控件中显示所有该驱动器下的文件。如果在目录的 ListBox 控件中对某个目录双击，则目录的 ListBox 控件将列出该目录下的文件夹，同时右边的 ListBox 控件列出该目录下的文件。在窗体上添加一个 ComBox 控件和 2 个 ListBox 控件，以合适的名字为这 3 个控件的 Name 属性命名。将 ComBox 控件的 DropDownStyle 属性设置为 DropDownList。Label 控件对每个控件的用途加以说明。由于需要在程序启动后便在 Combox 中列出计算机中的所有驱动器，因此对 Form 窗体的 Load 事件编写代码，该事件在窗体加载时被触发。接着为 ComBox 的 SelectedIndexChange 和 ListBox 的双击事件编写代码。整个程序的代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace CSHARP13_4
{
    public partial class Form1 : Form
    {
        private string[] directory;                //当前 ListBox 里显示的文件夹
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender,EventArgs e)
        {
            foreach(var logicDriver in Directory.GetLogicalDrives())
            {
                driverComboBox.Items.Add(logicDriver);
            }
        }

        private void DriverComboBox_SelectedIndexChanged(object sender,
              EventArgs e)
        {
            //得到某个驱动器下的所有文件夹
            directory=Directory.GetDirectories(driverComboBox.Text);
            directoryListBox.Items.Clear();
            foreach(var dir in directory)
            {
                //去掉路径名,仅显示文件夹的名字
```

```
31:                 directoryListBox.Items.Add
                    (Path.GetFileNameWithoutExtension(dir));
32:             }
33:             fileListBox.Items.Clear();
34:             foreach(var file in Directory.GetFiles(driverComboBox.Text))
35:             {
36:                 //去掉路径名,仅显示文件的名字
37:                 fileListBox.Items.Add(Path.GetFileName(file));
38:             }
39:         }
40:
41:         private void directoryListBox_DoubleClick(object sender,
                EventArgs e)
42:         {
43:             if (directory.GetUpperBound(0)==-1)
44:                 return;                     //ListBox 中没有项目
45:             //得到双击文件夹中包含的文件夹
46:             var currentDirectory=directory[directoryListBox.SelectedIndex];
47:             directory=Directory.GetDirectories(currentDirectory);
48:             directoryListBox.Items.Clear();
49:             foreach (var dir in directory)
50:             {
51:                 //去掉路径名,仅显示文件夹的名字
52:                 directoryListBox.Items.Add
                    (Path.GetFileNameWithoutExtension(dir));
53:             }
54:             fileListBox.Items.Clear();
55:             foreach (var file in Directory.GetFiles(currentDirectory))
56:             {
57:                 //去掉路径名,仅显示文件的名字
58:                 fileListBox.Items.Add(Path.GetFileName(file));
59:             }
60:         }
61:     }
62: }
```

程序的运行界面如图 13-3 所示。

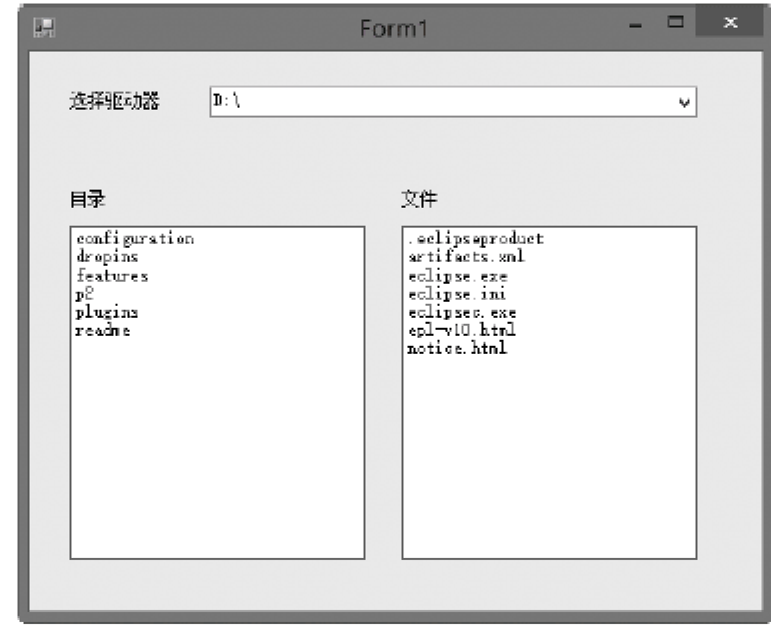


图 13-3 程序的运行界面

Directory 类的 GetLogicalDrives 方法返回所有的驱动器列表(第 17 行),将结果放入到 Combox 控件中显示。选择一个驱动器后,需要列出该驱动器下的文件夹和文件。通过调用 GetDirectories 方法得到该驱动器下所有文件夹的路径(第 26 行),该方法的返回结果是每一个文件夹的完整路径。而在

ListBox 中只显示文件夹名，因此在显示之前，对返回结果做了裁减(第 31 行)。由于文件夹完整的路径还需要使用，因此将它们存放在一个成员变量 dircetory 中(第 9 行)。

13.4.2 文件操作

和目录操作类似，静态的 File 类提供了对文件操作的方法，可以方便地创建、删除、移动或复制文件。以下是常用的一些方法：

- Copy(string sourceFileName, string destFileName)：sourceFileName 指出要复制文件的文件名以及该文件的路径，destFileName 指出新的副本的文件名可以带有路径。但 destFileName 不能是一个目录或者一个已存在的文件。
- Delete(string path)：删除一个文件。path 指出该文件名和路径。
- Exists(string path)As Boolean：测试 path 指定的文件是否存在。若存在，则返回 true，否则返回 false。
- Move(string sourceFileName, string destFileName)：将文件由位置 sourceFileName 处移动到新位置 destFileName 处。
- GetCreationTime(string path As String)：返回由 path 所指定文件的创建日期和时间。类似地还有 GetLastAccessTime，返回上次访问指定文件的日期和时间。GetLastWriteTime 返回上次写入指定文件或目录的日期和时间。

这些方法都是静态的，使用方法和 Directory 类相似。此外，如果需要对 Path 路径做一些处理，Path 类提供的常用方法有：

- ChangeExtension：更改路径字符串的扩展名。
- Combine(String[])：将字符串数组组合成一个路径。
- GetDirectoryName：返回指定路径字符串的目录信息。
- GetExtension：返回指定的路径字符串的扩展名。
- GetFileName：返回指定路径字符串的文件名和扩展名。
- GetFileNameWithoutExtension：返回不具有扩展名的指定路径字符串的文件名。
- GetFullPath：返回指定路径字符串的绝对路径。
- GetTempFileName：创建磁盘上唯一命名的零字节的临时文件，并返回该文件的完整路径。
- GetTempPath：返回当前用户的临时文件夹的路径。

例 13-5 批量文件改名。下面的程序通过字符串替换的方法，批量更改文件名。程序界面如图 13-4 所示。

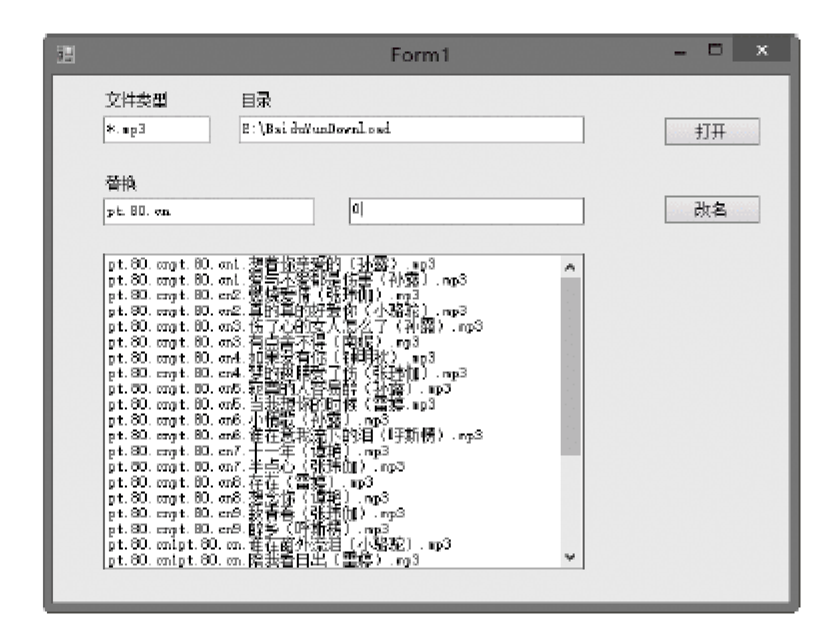


图 13-4 批量改名程序界面

在该程序中，在文件类型下输入要寻找的文件。单击“打开”按钮，将显示一个文件夹浏览对话框。选择一个文件夹，ListBox 中列出所有该类型的文件。单击“改名”按钮，将在所有文件的文件名中，匹配前一个文本框的字符，然后用后一个文本框的字符替换。代码如下：

```
using System;
using System.Windows.Forms;
using System.IO;

namespace replace
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        string[] fileNames;
        private void getButton_Click(object sender,EventArgs e)
        {
            FolderBrowserDialog fbd= new FolderBrowserDialog();
            if(fbd.ShowDialog()==DialogResult.OK)
            {
                dirTextBox.Text= fbd.SelectedPath;
            }
            fileNames= Directory.GetFiles(fbd.SelectedPath,
                    FliterTextBox.Text);
            foreach(var fileName in fileNames)
            {
                fileNameListBox.Items.Add(Path.GetFileName(fileName));
            }
        }

        private void replaceRenameButton_Click(object sender,EventArgs e)
        {
            fileNameListBox.Items.Clear();
            foreach(var fileName in fileNames)
            {
                try
                {
                    var newFileName= fileName.Replace
                        (replaceSoureTextBox.Text,
                            replaceDestTextBox.Text);
                    File.Copy(fileName,newFileName);
                    File.Delete(fileName);
                    fileNameListBox.Items.Add(Path.GetFileName
                        (newFileName));
                }
                catch { }
```

```
42:                 }
43:             }
44:         }
45: }
```

注意：File 类中没有 Rename 的方法，所以用 Copy＋Delete 来实现 Rename(第 37、38 行)。

习题

1. 假设有要排序的 20 个数存在文件 Data.txt 中。编写程序，打开该文件并将排好序的数重新写回该文件。

2. 重新打开第 1 题创建的文件，在文件的结尾再添加 10 个随机数。

3. 参考 Windows 的记事本程序，编写一个简单的文本编辑器程序。

4. 编写程序，在用户选择一个目录后，找出该目录及其子目录中所有后缀名为 doc 和 docx 的文件。

5. 假设有文本文件 1.txt 和 2.txt。编写程序，创建一个新的文本文件将 1.txt 中的内容和 2.txt 中的内容重复两遍，交替写入新的文本文件并删除 1.txt 和 2.txt。

6. 现代生活中有很多垃圾邮件。寻找 30 份垃圾邮件作为样本，将其保存为文本格式。扫描这 30 个文件中的词，统计出词频出现最高的 30 个词。另外随意挑选一些邮件，保存为文本格式，扫描这些邮件，根据它们含有上述 30 个词的多少给出一个打分的规则。最后根据分数，判断一个邮件是否是垃圾邮件。(注：中文的分词较难些，可以选择英文练习。)

第14章 数据结构



14.1 数据与数据结构

14.1.1 数据

什么叫**数据**？数据是描述客观事物的信息符号的集合，这些信息符号能被输入到计算机中存储起来，又能被程序处理、输出。事实上，数据这个概念本身是随着计算机的发展而不断扩展的概念。在计算机发展的初期由于计算机主要用于数值计算，数据指的就是整数、实数等数值；在计算机用于文字处理时，数据指的就是由英文字母和汉字组成的字符串；随着计算机硬件和软件技术的不断发展，扩大了计算机的应用领域，诸如表格、图形、图像、声音等也属于数据的范畴。目前非数值问题的处理占用90%以上的计算机时间。

数据类型是程序设计中的概念，程序中的数据都属于某个特殊的数据类型，它是指具有相同特性的数据的集合。数据类型决定了数据的性质，如取值范围、操作运算等。常用的数据类型有整型、浮点型、字符型等。数据类型还决定了数据在内存中所占空间的大小，如字符型占一个字节，而长整型一般占4个字节等。

对于复杂一些的数据，仅用数据类型无法完整地描述。如表14-1所示，表示教师得分要描述教师的姓名及各项得分，这时需要用到数据元素的概念。教师得分登记表的数据元素是姓名、教学得分、科研得分、其他得分、合计，也就是说每个数据元素由姓名、教学得分、科研得分、其他得分、合计五个数据项组成。这五个数据项含义明确，若再细分就无明确独立的含义，属于基本数据类型（字符型和整型或浮点型）。数据元素中可能用到多个数据类型（称为数据项），共同描述一个客体，如教师。数据元素有时也被称为记录或结点。在程序设计中，前面所说的数据类型又被称为基本数据类型，由基本数据类型组成的数据元素的定义被称为构造数据类型（结构和类都属此列）。

表14-1　教师得分登记表

姓　名	教学得分	科研得分	其他得分	合计
张力	35	34	11	80
王五	36	35	12	83
…	…	…	…	…

14.1.2 数据结构

计算机的处理效率与数据的组织形式和存储结构密切相关。这类似于人们所用的《英语词典》、《科学技术辞海》和《新华字典》等工具书，它们都是按英文字母或拼音字母的顺序组织排列“词条”，这样人们查阅工具书的速度较快。假如“词条”不是按字母顺序组织排列，而是任意组织排列，那么查词速度一定很慢。因此，很有必要研究数据的组织形式和存储结构，而且在当今网络世界中传递数据更加依赖于数据的组织形式和存储结构。

什么是**数据结构**？数据结构在计算机科学界至今没有标准的定义，根据不同的理解有不同的表述方法。

Sartaj Sahni 在其《数据结构、算法与应用》一书中称：“数据结构是数据对象以及存在于该对象的实例和组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。”他将数据对象(data object)定义为“一个数据对象是实例或值的集合”。

Clifford A. Shaffer 在《数据结构与算法分析》一书中的定义是：“数据结构是抽象数据类型(Abstract Data Type，ADT)的物理实现。”

Lobert L. Kruse 在《数据结构与程序设计》一书中，将一个数据结构的设计过程分成抽象层、数据结构层和实现层。其中，抽象层是指抽象数据类型层，它讨论数据的逻辑结构及其运算，数据结构层和实现层讨论一个数据结构的表示和在计算机内的存储细节以及运算的实现。

由此可见，在任何问题中，构成数据的数据元素并不是孤立存在的，它们之间存在着一定的关系以表达不同的事物及事物之间的联系。所以简单地说，数据结构就是研究数据及数据元素之间关系的一门学科，它包括三个方面的内容：

- 数据的逻辑结构；
- 数据的存储结构；
- 数据的运算(即数据的处理操作)。

一般认为，一个数据结构是由数据元素依据某种逻辑联系组织起来的。对数据元素间逻辑关系的描述称为数据的逻辑结构；数据必须在计算机内存储，数据的存储结构是数据结构的实现形式，是其在计算机内的表示；此外讨论一个数据结构必须同时讨论在该类数据上执行的运算才有意义。

1. 数据的逻辑结构

数据的逻辑结构就是数据元素之间的逻辑关系。这里，我们对数据所描述的客观事物本身的属性意义不感兴趣，只关心它们的结构及关系。将那些在结构形式上相同的数据抽象成某一数据结构，比如线性表、树和图等。

根据数据元素之间关系的不同特性，数据结构又可分为以下四大类(见图 14-1)：

(1) 集合：数据元素之间的关系只有“是否属于同一个集合”。

(2) 线性结构：数据元素之间存在线性关系，即最多只有一个前导和后继元素。

(3) 树形结构：数据元素之间为层次关系，即最多有一个前导和多个后继元素。

（4）图状结构：数据元素之间的关系为多对多的关系。

其中树和图又被统称为非线性数据结构。

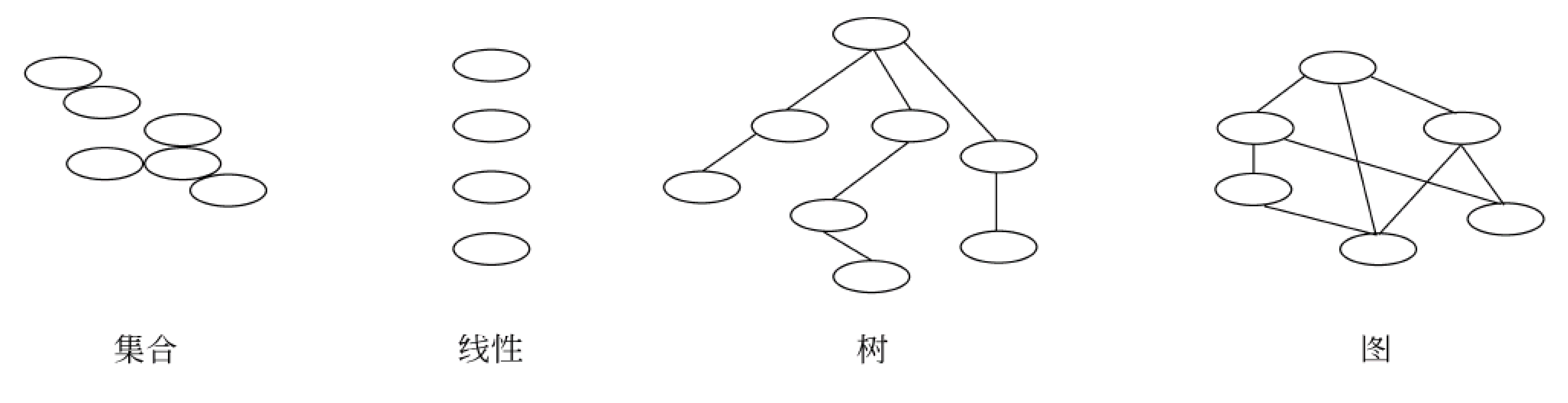


图 14-1 四种逻辑结构示意图

2. 数据的存储结构

数据的逻辑结构是从逻辑上来描述数据元素之间的关系的，是独立于计算机的。然而讨论数据结构的目的是为了在计算机中实现对它的处理，因此还需要研究数据元素和数据元素之间的关系如何在计算机中表示，这就是数据的存储结构，又称数据的映像。

计算机的存储器是由很多个存储单位组成的，每个存储单元有唯一的地址。数据的存储结构要讨论的就是数据结构在计算机存储器上的存储映像方法。根据数据结构的形式定义，数据结构在存储器上的映像，不仅包括数据元素集合如何存储映像，而且还包括数据元素之间的关系如何存储映像。

一般来说，数据在存储器中的存储有四种基本的映像方法。

（1）顺序存储结构：就是把数据元素按某种顺序放在一块连续的存储单元中，其特点是借助数据元素在存储器中的相对位置来表示数据元素之间的关系。顺序存储的问题是，如果元素集合很大，则可能找不到一块很大的连续的空间来存放。

（2）链式存储结构：有时往往存在这样一些情况：存储器中没有足够大的连续可用的空间，只有不相邻的零碎小块存储单元；另一种情况是在事前申请一段连续空间时，因无法预计所需存储空间的大小，需要临时增加空间。所有这些情况，要得到一块合适的连续存储单元并非易事，即这种情况下顺序存储结构无法实现。

链式存储结构的特点是将存放每个数据元素的结点分为两部分：一部分存放数据元素（称为数据域）；另一部分存放指示存储地址的指针（称为指针域），借助指针表示数据元素之间的关系。结点的结构如下：

数据域	指针域

链式存储结构可用一组任意的存储单元来存储数据元素，这组存储单元可以是连续的，也可以是不连续的。链式存储因为有指针域，增加了额外的存储开销，并且实现上也较为麻烦，但大大增加了数据结构的灵活性。

（3）索引存储结构：在线性表中，数据元素可以排成一个序列：R1，R2，…，Rn，每个数据元素 Ri 在序列里都有对应的位置码 i，这就是元素的索引号。索引存储结构就是通过数据元素的索引号 i 来确定数据元素 Ri 的存储地址。一般索引存储结构有两种实

现方法：①建立附加的索引表，索引表里第 i 项的值就是第 i 个元素的存储地址；②当每个元素所占单元数都相等时，可用位置码 i 的线性函数值来确定元素对应的存储地址，即

$$Loc(Ri)=(i-1)*L+d_0$$

（4）散列存储结构：这种存储方法就是在数据元素与其在存储器上的存储位置之间建立一个映像关系 F。根据这个映像关系 F，已知某数据元素就可以得到它的存储地址。即 D=F(E)，这里 E 是要存放的数据元素，D 是该数据元素的存储位置。可见，这种存储结构的关键是设计这个函数 F，但函数 F 不可能解决数据存储中的所有问题，还应有一套意外事件的处理方法，它们共同实现数据的散列存储结构。哈希表是一种常见的散列存储结构。

3. 数据的运算

数据的运算是定义在数据逻辑结构上的操作，如插入、删除、查找、排序、遍历等。每种数据结构都有一个运算的集合。

14.2 线性表

线性表是最基本、最简单也是最常用的一种数据结构。线性表中数据元素之间的关系是一对一的关系，即除了第一个和最后一个数据元素之外，其他数据元素都是首尾相接的。线性表的逻辑结构简单，便于实现和操作，在实际应用中是广泛采用的一种数据结构。

14.2.1 线性表的逻辑结构及运算

线性表是一个线性结构，它是一个含有 n≥0 个结点的有限序列，对于其中的结点，有且仅有一个开始结点（第一个结点）没有前驱，但有一个后继结点；有且仅有一个终端结点（最后一个结点）没有后继，但有一个前驱结点；其他的结点都有且仅有一个前驱和一个后继结点。

一般地，一个线性表可以表示成一个线性序列：k1，k2，…，kn，其中 k1 是开始结点，kn 是终端结点。线性表具有以下一些基本性质：

- 数据元素的个数 n 定义为表的长度。当 n=0 时，称为空表。空表中无数据元素。
- 若表非空，则必存在唯一的开始结点。
- 必存在唯一的终端结点。
- 除最后一个元素之外，其余结点均有唯一的后继。
- 除第一个元素之外，其余结点均有唯一的前驱。
- 数据元素 ki(1≤i≤n)在不同情况下的具体含义不同，它可以是一个数，或者是一个符号，或者是更复杂的信息。虽然不同数据表的数据元素可以是各种各样的，但对于同一线性表的各数据元素必定具有相同的数据类型和长度。

例 14-1 线性表的例子。

- 某班学生的数学成绩(78，92，66，84，45，72，92)是一个线性表，每个数据元素是一

个正整数，表长为 7。

- 一星期的七天的英文缩写词(SUN,MON,TUE,WED,THU,FRI,SAT)是一个线性表，表中数据元素是一个字符串，表长为 7。
- 某企业职工基本工资情况((张三，助工，3，543)，(李四，高工，21，986)，(王五，工程师，9，731))亦是一个线性表，表中数据元素是由姓名、职称、工龄，基本工资四个数据项组成的一个记录(对象)，表长为 3。

线性表可以进行的常用基本操作有以下几种：

(1) 置空表：将线性表 L 的表长置为 0。

(2) 求表长：求出线性表 L 中数据元素的个数。

(3) 取表中元素：仅当 1≤i≤Length(L)时，取得线性表 L 中的第 i 个元素 ki(或 ki 的存储位置)，否则无意义。

(4) 取元素 ki 的直接前趋：当 2≤i≤Length(L)时，返回 ki 的直接前趋 ki－1。

(5) 取元素 ki 的直接后继：当 1≤i≤Length(L)－1 时，返回 ki 的直接后继 ki＋1。

(6) 定位：返回元素 x 在线性表 L 中的位置。若在 L 中有多个 x，则只返回第一个 x 的位置；若在 L 中不存在 x，则返回 0。

(7) 插入：在线性表 L 的第 i 个位置上插入元素 x，运算结果使得线性表的长度增加 1。

(8) 删除：删除线性表 L 的第 i 个位置上的元素 ki，此运算的前提应是 Length(L)≠0，运算结果使得线性表的长度减 1。

对线性表还有一些更为复杂的操作，如：将两个线性表合并成一个线性表，将一个线性表分解为 n 个线性表，对线性表中的元素按值的大小重新排列等。这些运算都可以通过上述 8 种基本运算的组合派生来实现。

14.2.2 线性表的存储结构

要使线性表成为计算机可以处理的对象，就必须把线性表的数据元素及数据元素之间的逻辑关系都存储到计算机的存储器中。线性表常用顺序方式和链表方式来存储。

1. 线性表的顺序存储

线性表的顺序存储结构就是将线性表的每个数据元素按其逻辑次序依次存放在一组地址连续的存储单元里。由于逻辑上相邻的元素存放在内存的相邻单元中，所以线性表的逻辑关系蕴含在存储单元的物理位置相邻的关系中。也就是说，在顺序存储结构中，线性表的逻辑关系的存储是隐含的。

设线性表中每个元素占用 C 个存储单元，用 Loc(ki)表示元素 ki 的存储位置，则顺序存储结构的存储示意图如图 14-2 所示。

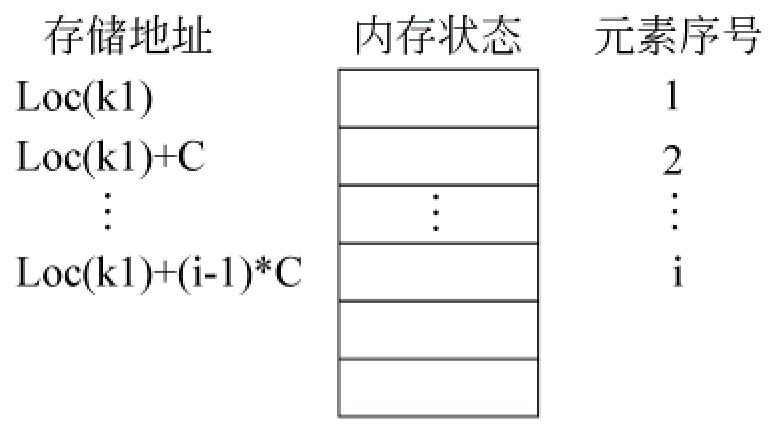


图 14-2 线性表的顺序存储结构

从图 14-2 中可以看出，若已知线性表的第一个元素的存储位置是 Loc(k1)，则第 i 个元素的存储位置为：

$$Loc(k_i)=Loc(k_1)+C*(i-1) \quad 1\leqslant i\leqslant n$$

可见,线性中每个元素的存储地址是该元素在表中序号的线性函数。只要知道某元素在线性表中的序号就可以确定其在内存中的存储位置。所以说,线性表的顺序存储结构是一种随机存取结构。

数组是在内存中连续分配的。所以数组天生就是一种线性结构。用数组来实现线性表,可以预先定义一个较大的数组,用来存放线性表中的元素。元素从数组的 0 位置存起,数组最后的一些位置是空闲的。

例 14-2 一个整数线性表的实现。用整型数组存储元素,实现线性表的基本操作。使用数组 list 来存储元素。list 的大小设为 MAX=1000。表长用 n 来表示,线性表中的每一个元素都是整数。这里,使用一个类来描述线性表。

程序代码如下:

```
using System;

namespace CSHARP14_2
{
    class MyList
    {
        private int[] list;
        private int n;
        private const int MAX=1000;

        public MyList()
        {
            list=new int[MAX];
        }

        public MyList(int length)
        {
            list=new int[length];
        }
        ///<summary>
        ///置空表
        ///</summary>
        public void Initiate()
        {
            n=0;
        }
        ///<summary>
        ///求表长
        ///</summary>
        ///<returns>表的长度,0表示空表</returns>
        public int GetLength()
```

```
        {
            return n;
        }
        ///< summary>
        ///得到表中的第 i 个元素的值
        ///< /summary>
        ///< param name= "i">索引< /param>
        ///< returns>第 i 个元素的值,索引超出范围则抛出异常< /returns>
        public int GetData(int i)
        {
            if (i >= 0 && i< n)
                return list[i];
            else
                throw new Exception("索引超出范围");
        }
        ///< summary>
        ///求第 i 个元素的直接前趋
        ///< /summary>
        ///< param name= "i">第 i 个元素< /param>
        ///< returns>第 i-1 个元素的值,如果超出范围则抛出异常< /returns>
        public int PriorData(int i)
        {
            if (i >= 1 && i< n)
                return list[i - 1];
            else
                throw new Exception("索引超出范围");
        }
        ///< summary>
        ///求第 i 个元素的直接后继
        ///< /summary>
        ///< param name= "i">第 i 个元素< /param>
        ///< returns>第 i+1 个元素的值,如果超出范围则抛出异常< /returns>
        public int NextData(int i)
        {
            if (i >= 0 && i< n - 1)
                return list[i+1];
            else
                throw new Exception("索引超出范围");
        }
        ///< summary>
        ///返回 x 在表中的位置
        ///如果有多个 x,则返回第一个 x 的位置;若 x 不存在,则返回 -1
        ///< /summary>
        ///< param name= "x">需要定位的元素< /param>
```

```
        ///<returns>元素位置</returns>
        public int Locate(int x)
        {
            int i=0;
            while(i<n)
            {
                if (x==list[i])
                    return i;
                i++;
            }
            return -1;
        }
        ///<summary>
        ///在表中插入元素
        ///</summary>
        ///<param name="x">要插入的元素</param>
        ///<param name="i">插入的位置</param>
        public void Insert(int x,int i)
        {
            if(i>=0 && i<=n)
            {
                //移动表中元素
                var j=n;
                while(j>i)
                {
                    list[j]=list[j-1];
                    j--;
                }
                //插入 x
                list[i]=x;
                n++;
               }
            else
            {
                throw new Exception("插入的位置不正确");
            }
        }
        public void Delete(int i)
        {
            if(i>=0 && i<n)
            {
                //i+1以后的元素依次前移
                while (i<n)
                {
```

```
                        list[i]= list[i+ 1];
                        i++;
                    }
                    n--;
                }
                else
                {
                    throw new Exception("删除的位置不正确");
                }
            }
        }
}
```

需要注意的是：由于数组下标是从 0 开始的，为方便起见，本程序实现的线性表默认的第一个元素下标是 0。因此在第 5 个位置上插入，实际是在线性表的第 6 个位置上插入。也可以修改程序，使其和前面描述的线性表一致。

不需要界面，建立一个控制台程序，对上面的类进行测试，代码如下：

```
using System;

namespace CSHARP14_2
{
    class Program
    {
        static void Main(string[] args)
        {
            //线性表测试
            MyList myList= new MyList();
            myList.Initiate();          //置空表
            //插入 10 个整数
            for (int i= 0; i< 10; i++)
                myList.Insert(i+ 1,i);
            Print(myList);
            //在位置 5 插入 99
            myList.Insert(99,5);
            Print(myList);
            //删除位置 8 的数
            myList.Delete(8);
            Print(myList);
        }
        static void Print(MyList list)
        {
            for (int i= 0;i< list.GetLength();i++)
            {
```

```
                    Console.Write(list.GetData(i).ToString()+" ");
                }
                Console.WriteLine(" 表长：{0}",list.GetLength());
            }
        }
}
```

程序的运行结果如下：

```
1 2 3 4 5 6 7 8 9 10   表长：10
1 2 3 4 5 99 6 7 8 9 10   表长：11
1 2 3 4 5 99 6 7 9 10   表长：10
```

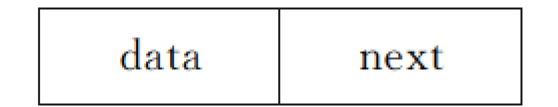
2. 线性表的单链表存储

线性表的顺序存储结构是把整个线性表存放在一片连续的存储区，其逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，每个元素的存储位置可用一个简单、直观的公式来表示。然而，某一线性表中的元素频繁进行插入和删除操作时，为了保持元素在存储区的连续性，在插入元素时必须移动大量元素给新插入的元素"腾位置"；而在删除时，又必须移动大量后继元素"补缺"，因而在操作执行时要花大量时间去移动数据元素。

能否设计一种新的存储结构，在元素插入、删除时无须改变已存储元素的位置？这就是我们将讨论的另一种存储结构——链式存储结构。

用链式方式存储一个线性表，其特点是用一组任意的存储区存储该线性表，此存储区可以是连续的，也可以是分散的。这样，逻辑上相邻的元素在物理位置上就不一定是相邻的，为了能正确反映元素的逻辑顺序，就必须在存储每个元素 ai 的同时，存储其直接后继（或直接前趋）的存储位置。

链式存储方式由很多种，我们在这里仅介绍单链表。在单链表中每个结点都由两部分组成：存储数据元素的数据域（data）；存储直接后继结点存储位置的指针域（next）。其结点结构如下：



一个由学生姓名组成的线性表（张三，李四，王五，赵六）采用单链表为存储结构时，其单链表的逻辑结构如图 14-3 所示。

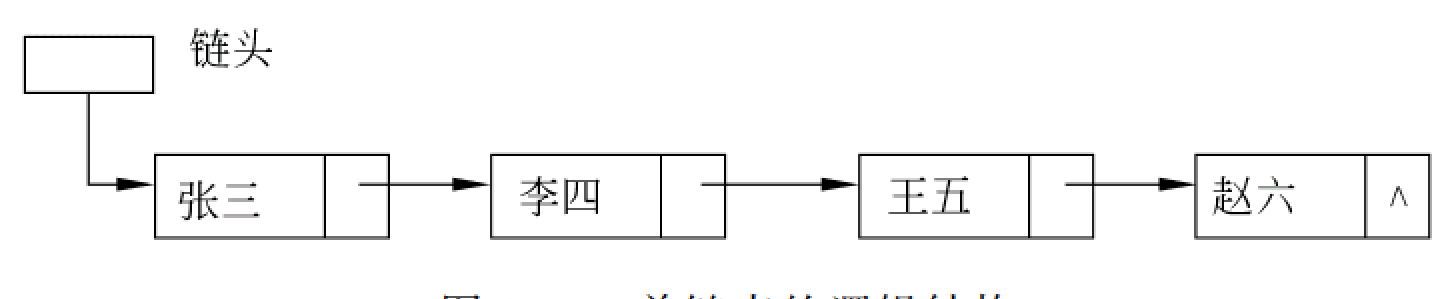


图 14-3　单链表的逻辑结构

在图 14-3 所示的单链表中，链头是指向单链表中第一个结点的指针，称之为头指针；最后一个元素赵六所在结点不存在后继，因而其指针域为"空"（用 NULL 或 ∧ 表示）。该

单链表在存储区的物理状态如图 14-4 所示。

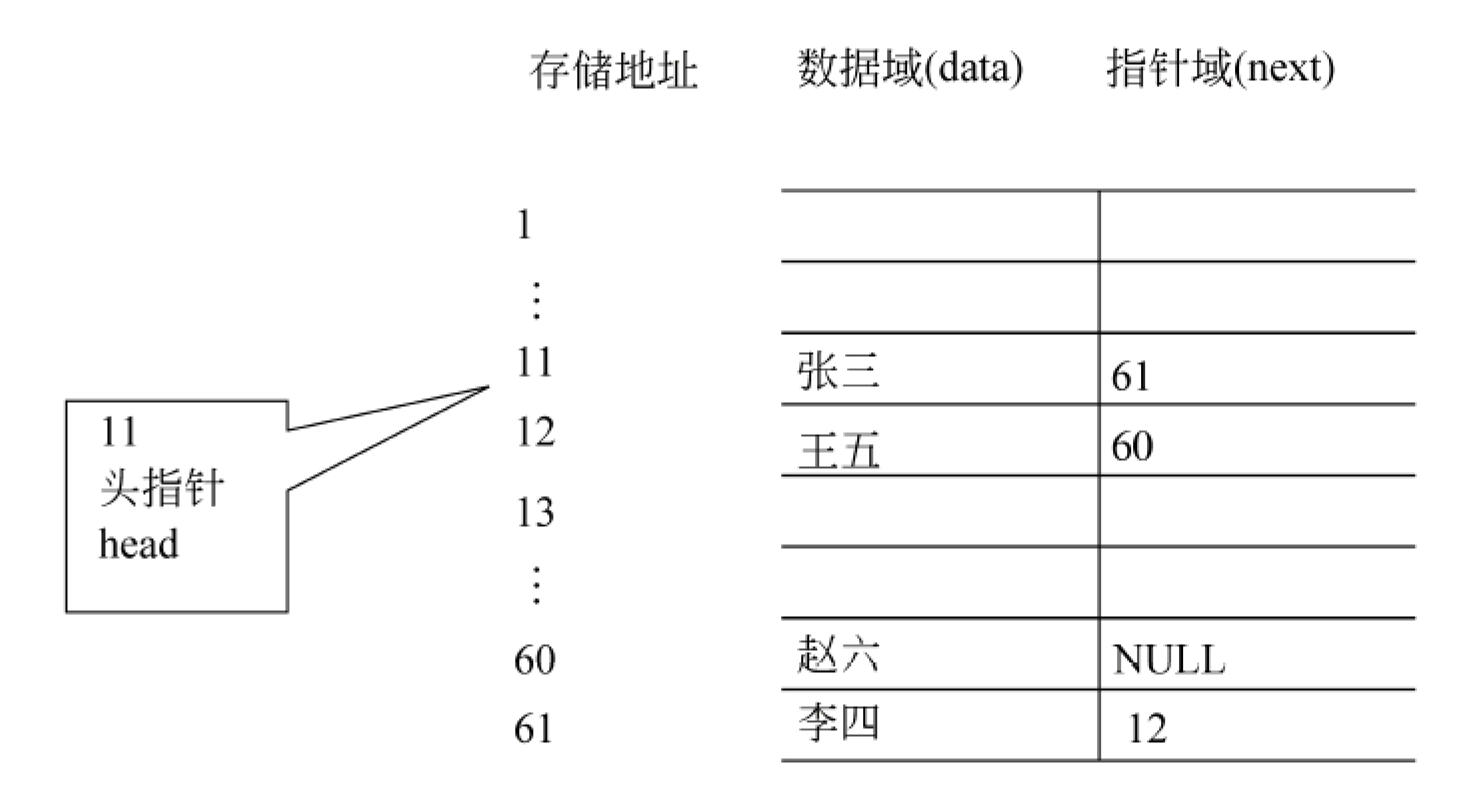


图 14-4　单链表在存储区的物理状态

在单链表存储结构下，顺序表的插入和删除这两个操作的实现方法如下。

1）单链表的插入

设有线性表(a_1,a_2,…,a_i,a_{i+1},…,a_n)，用单链表存储，头指针为 head，要求在存储数据元素 a_i 的结点之前插入一个数据元素为 X 的结点。设新插入的结点指针是 S。

若已知 a_i 的前趋 a_{i-1}所在结点的指针 P，只要执行以下两步操作即可：

step1　令结点 S 的指针域指向 a_i 所在的结点(S→next＝P→next)；

step2　令结点 P 的指针域指向结点 S(P→next＝S)。

执行插入后的单链表的逻辑状态如图 14-5 所示。

由此可见，插入操作执行之前，首先就是要找到单链表中插入位置的前一个结点的指针(存储位置)。由于知道头指针，因此可以从头指针一一找到下一个元素，直到所需的元素。与顺序存储方式的随机访问相比，这是链式存储的不便之处。

2）单链表的删除

删除操作和插入操作一样，首先要搜索单链表以找到指定删除结点的前趋结点(假设为 P)，然后只要将待删除结点的指针域内容赋予 P 结点的指针域就可以了。删除元素所在的结点之后，单链表的逻辑状态如图 14-6 所示。



图 14-5　在带头结点的单链表中插入结点 S　　图 14-6　从带头结点的单链表中删除一个结点

14.2.3　List 类

在 Visual C＃中，除了使用类自己编程实现线性表外，还可以通过.NET 提供的 List 类来实现。按照使用类的方法，应该先声明一个 List 类的对象，这样就有了一个线性表对象。需要注意的是 List 类是一个泛型类。泛型是一个较复杂的概念，在此，可简单理

解为创建一个 List 对象时需要指明将在 List 中存储何种类型的数据。例如，声明一个整数类型的 List 对象的格式如下：

```
List< int> list= new List< int> (1000);
```

例 14-3 使用 List 类实现例 14-2。

代码如下：

```
using System;
using System.Collections.Generic;

namespace CSHARP14_3
{
    class Program
    {
        static void Main(string[] args)
        {
            List< int> list= new List< int> (1000);
            //插入 10个元素并显示
            for (int i= 0; i< 10; i+ + )
                list.Insert(i,i+ 1);
            Print(list);
            //在位置 5插入 99并显示
            list.Insert(5,99);
            Print(list);
            //删除第 8个元素
            list.RemoveAt(8);
            Print(list);
            //排序
            list.Sort();
            Print(list);
            //反转
            list.Reverse();
            Print(list);
        }
        static void Print(List< int> l)
        {
            for (int i= 1; i< l.Count; i+ + )
                Console.Write(l[i].ToString()+ " ");
            Console.WriteLine("表长为: {0}",l.Count);
        }
    }
}
```

程序的运行结果如下：

```
2 3 4 5 6 7 8 9 10 表长为: 10
2 3 4 5 99 6 7 8 9 10 表长为: 11
2 3 4 5 99 6 7 9 10 表长为: 10
2 3 4 5 6 7 9 10 99 表长为: 10
10 9 7 6 5 4 3 2 1 表长为: 10
```

可以看到程序的运行结果和例 14-2 基本相似,但有一些区别。List 类并没有提供前驱或者后继方法,但这通过下面罗列的 List 类的其他方法很容易实现。其次,List 类提供了排序方法,调用该方法可以对 List 中的元素排序。但是需要注意的是,在排序时,程序要知道如何比较 List 中 2 个元素的大小。由于可以将自定义的数据类型存储到 List 中,此时程序就不清楚如何比较大小了,因而也无法排序(或者说排序需要使用者提供一个比较的方法,可使用 C# 的委托,这已超出了本书的讨论范围)。List 类中常用的方法如下:

- Add:将对象添加到 List 的结尾处。
- Clear:从 List 中移除所有元素(置空表)。
- Contains:确定某元素是否在 List 中。如在,Contains 返回 True,否则返回 False。
- FindIndex:搜索与指定条件相匹配的元素,返回 List 或它的一部分中第一个匹配项的从零开始的索引。
- IndexOf:返回 List 或它的一部分中某个值的第一个匹配项的从零开始的索引。
- Insert:将元素插入 List 的指定索引处。
- LastIndexOf:返回 List 或它的一部分中某个值的最后一个匹配项的从零开始的索引。
- Remove:从 List 中移除特定对象的第一个匹配项。
- RemoveAt:移除 List 的指定索引处的元素。
- Reverse:将 List 或它的一部分中的元素的顺序反转。
- Sort:对 List 或它的一部分中的元素进行排序。

List 中最常用的属性只有一个 List.Count,表示 List 表中的元素个数(表长)。

14.2.4 LinkedList 类

和 List 类一样,.NET 提供了 LinkedList 类用以帮助实现链表。LinkedList 类通常还需要和 LinkedListNode 类一起使用,LinkedListNode 类表示链表中的一个结点。本节以一个简单的例子来说明,读者若有兴趣请在微软的网站(MSDN)上搜索资料。

例 14-4 对图 14-4 所示的链表做一个简单的实现。图 14-4 的链表中每一个结点存储的数据类型是一个字符串。先生成一个空的链表,然后用所给的名字创建每一个结点,并加入到链表中。程序中还编写了一个 Display 过程用于显示链表的内容。代码如下:

```
using System;
using System.Collections.Generic;

namespace CSHARP14_4
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            //定义一个空的链表
            LinkedList<string> linkedList= new LinkedList<string>();
            //生成链表的4个结点
            LinkedListNode<string> linkedListNodeA=
                new LinkedListNode<string>("张三");
            LinkedListNode<string> linkedListNodeB=
                new LinkedListNode<string>("李四");
            LinkedListNode<string> linkedListNodeC=
                new LinkedListNode<string>("王五");
            LinkedListNode<string> linkedListNodeD=
                new LinkedListNode<string>("赵六");

            //将结点加入到链表中,实际运用中不一定要依照下面的顺序加入。
            //在此,为了演示AddFirst、AddLast、AddAfter和AddBefore的用法
            //而这么做,在首尾加入链表时(调用AddFirst和AddLast),
            //也可以不声明LinkedListNode对象,而直接加入,如:
            //' linkedListA.AddFirst("张三")
            linkedList.AddFirst(linkedListNodeA);
            linkedList.AddLast(linkedListNodeD);
            linkedList.AddAfter(linkedListNodeA,linkedListNodeB);
            linkedList.AddBefore(linkedListNodeD,linkedListNodeC);

            Display(linkedList);

            //移除第一个结点
            linkedList.RemoveFirst();
            Display(linkedList);

            //移除结点C
            linkedList.Remove(linkedListNodeC);
            Display(linkedList);
        }

        static void Display(LinkedList<string> link)
        {
            foreach (var name in link)
                Console.Write(name+" ");
            Console.WriteLine("表长为:{0}",link.Count);
        }
```

```
49:        }
50: }
```

程序的运行结果：

```
张三 李四 王五 赵六 表长为:4
李四 王五 赵六 表长为:3
李四 赵六 表长为:2
```

14.3　栈和队列

栈和队列也是线性结构，线性表、栈和队列这三种数据结构的数据元素以及数据元素间的逻辑关系完全相同，差别是线性表的操作不受限制，而栈和队列的操作受到限制。栈的操作只能在表的一端进行，队列的插入操作在表的一端进行而其他操作在表的另一端进行，所以把栈和队列称为操作受限的线性表。

14.3.1　栈

栈是只能在某一端插入和删除的特殊线性表。它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶(top)，另一端称为栈底(bottom)；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈(push)，删除则称为出栈(pop)。栈也称为先进后出表。

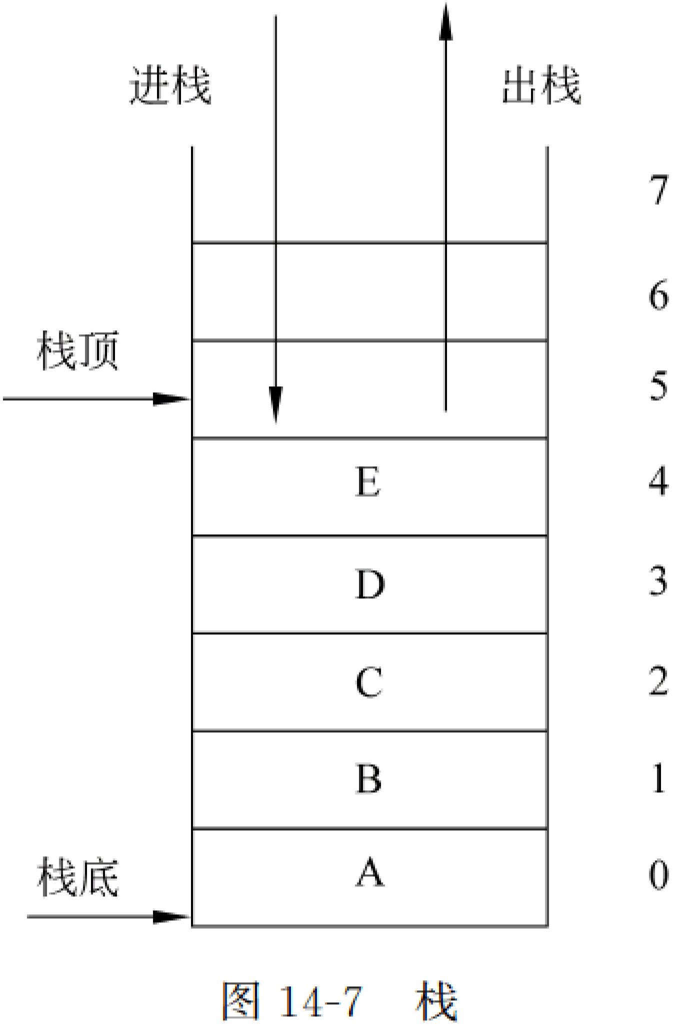


图 14-7　栈

栈顶始终指向栈顶最后一个元素之后的空位置。在图 14-7 中，栈里面共有 5 个元素，入栈的次序依次是 ABCDE。栈底始终等于 0，而栈顶等于 5。图 14-8 描述了最后 2 个元素出栈、F 进栈的情形。

图 14-8(a)最后一个元素 E 出栈，栈顶＝4；图 14-8(b) D 出栈，栈顶＝3；图 14-8(c)元素 F 进栈，栈顶＝4。

当栈中没有元素的时候，称为空栈，空栈的条件是栈顶＝栈底。栈的大小一般是预先定义好的，当栈顶＝栈的大小时，称为栈满。显然，当栈为空的时候，不能进行出栈操作；而当栈满的时候，不能进行入栈操作。通常对栈有如下几个操作：

- 求栈的长度：GetLength，返回栈中数据元素的个数。
- 判断栈是否为空：IsEmpty，如果栈为空返回 true，否则返回 false。
- 清空栈：Clear，使栈为空。
- 入栈操作：Push，将新的数据元素添加到栈顶，栈发生变化。
- 出栈操作：Pop，将栈顶元素从栈中取出，栈发生变化。

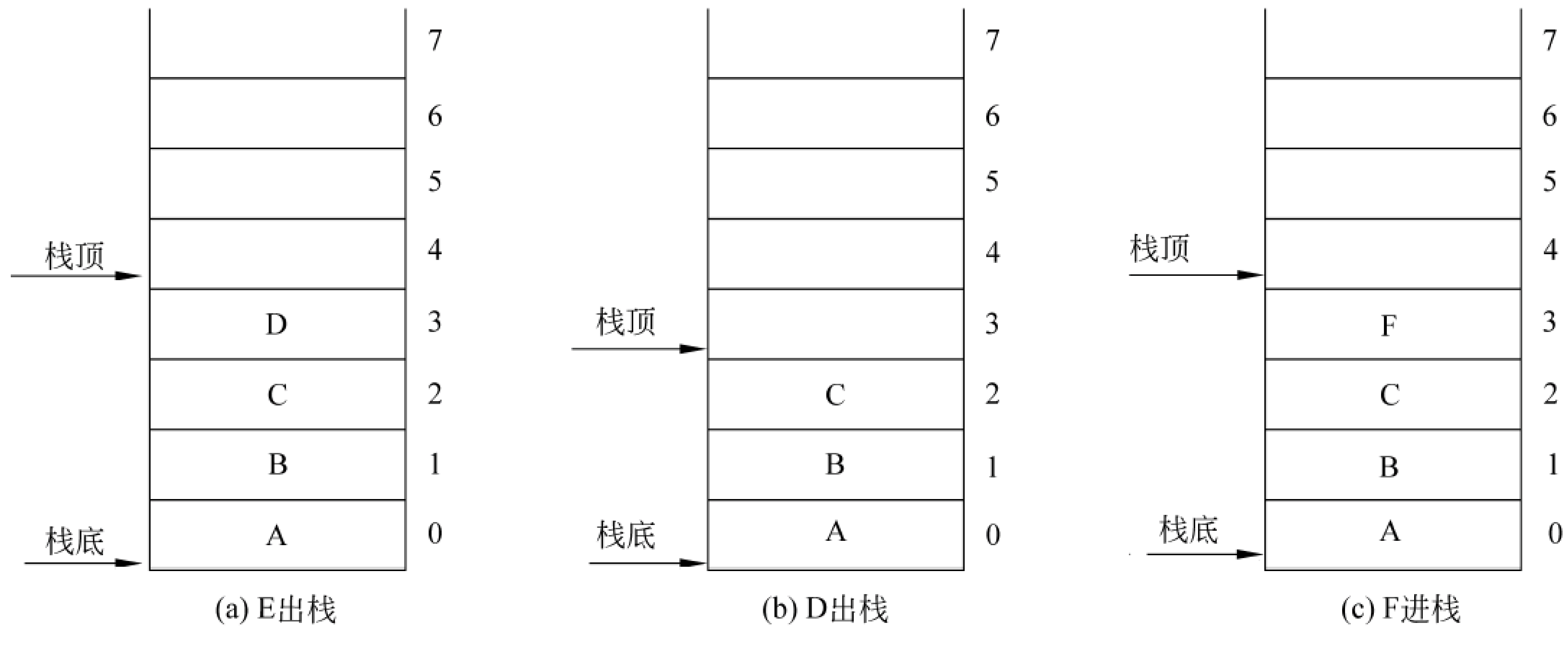


图 14-8　栈的变化

- 取栈顶元素：GetTop，返回栈顶元素的值，栈不发生变化。

同样，栈在计算机中的存储结构也有顺序存储和链式存储两种，显然顺序结构可以用一个数组来实现。

例 14-5　实现一个字符串的栈。用一个指定大小的数组来存储栈的内容。在此，假设栈里存储的是字符串。变量 top 保存栈顶的数组下标，变量 bottom 为栈底，始终为 0。

程序代码如下：

```
using System;

namespace CSHARP14_5
{
    class MyStack
    {
        private string[] stackData;
        private int top=0;
        private int buttom=0;

        public MyStack()
        {
            stackData=new string[100];
        }
        public MyStack(int n)
        {
            stackData=new string[n];
        }

        ///<summary>
        //清空栈，也就是将栈顶和栈底指针设为 0
        ///</summary>
```

```
        public void Clear()
        {
            top=buttom=0;
        }
        ///<summary>
        ///入栈,存入数组中,同时栈顶加一
        ///</summary>
        ///<param name="item">入栈的元素</param>
        public void Push(string item)
        {
            if(top<stackData.Length-1)
            {
                stackData[top]=item;
                top++;
            }
            else
            {
                throw new Exception("Stack Full");
            }
        }
        ///<summary>
        ///检查栈是否为空
        ///</summary>
        ///<returns>是,true,否,false</returns>
        public bool IsEmpty()
        {
            return top==buttom;
        }
        ///<summary>
        ///出栈函数
        ///</summary>
        ///<returns>返回栈顶的元素,若空则抛出异常</returns>
        public string Pop()
        {
            if (!IsEmpty())
            {
                top--;
                return stackData[top];
            }
            else
                throw new Exception("Stack Empty");
        }
        ///<summary>
        ///得到栈中元素的个数
```

```
            ///</summary>
            ///<returns>栈中的元素个数</returns>
            public int GetLength()
            {
                return top - buttom;
            }
        }
}
```

测试代码如下：

```
using System;

namespace CSHARP14_5
{
        class Program
        {
            static void Main(string[] args)
            {
                MyStack myStack= new MyStack();
                //初始栈,置为空栈
                myStack.Clear();
                //将一些字符串压入栈
                myStack.Push("西安");
                myStack.Push("交通");
                myStack.Push("大学");

                //只要栈不空,将栈里的字符串全部弹出栈并显示
                while (!myStack.IsEmpty())
                    Console.Write(myStack.Pop()+" ");
                Console.WriteLine();
                }
        }
}
```

程序的运行结果如下：

```
大学 交通 西安
```

14.3.2 Stack 类

.NET 提供了 Stack 类来完成栈的运算。和前面的 List 类相似，Stack 类是面向对象的泛型类，可以在声明的时候指定栈中的数据类型。Stack 类主要提供了以下方法：

- Clear：从 Stack 中移除所有对象。
- Contains：确定某元素是否在 Stack 中。
- Peek：返回位于 Stack 顶部的对象但不将其移除。

- Pop：移除并返回位于 Stack 顶部的对象。
- Push：将对象插入 Stack 的顶部。

它还有一个重要的属性 Count 指出栈中元素的个数。注意：没有一个单独的判断栈是否为空的方法，可以通过 Count 属性是否大于 0 来判断。

例 14-6 使用 Stack 类实现例 14-5。程序代码如下：

```
using System;
using System.Collections.Generic;

namespace CSHARP14_6
{
    class Program
    {
        static void Main(string[] args)
        {
            //声明一个栈
            Stack<string> stack=new Stack<string>();
            //初始栈,置为空栈
            stack.Clear();
            //将一些字符串压入栈
            stack.Push("西安");
            stack.Push("交通");
            stack.Push("大学");
            //只要栈不空,将栈里的字符串全部弹出栈并显示
            while (stack.Count > 0)
                Console.Write(stack.Pop()+" ");
            Console.WriteLine();
        }
    }
}
```

程序的运行结果和例 14-5 是一样的，只是使用了.NET 的 Stack 类。

例 14-7 检查表达式的括号是否匹配。从键盘输入一个表达式如 (a+b)*(5+c)*((22-c)/23+56)，现在要检查输入的表达式中括号是否匹配，这可以用栈来实现。从左至右逐一读取表达式中的每一个字符。如果是左括号"("，则将其压入栈中；如果遇到一个右括号")"，则从栈中弹出一个左括号。当处理完表达式的字符串时，如果栈恰好也是空的，则表达式是匹配的。否则，如果处理完表达式栈不空；或表达式未处理完，需要出栈时栈是空的；则可以断定，括号是不匹配的。

程序代码如下：

```
using System;
using System.Collections.Generic;

```

```
namespace CSHARP14_7
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("请输入表达式");
            var expression=Console.ReadLine();
            //声明一个栈
            Stack<char> stack=new Stack<char>();
            //初始栈,置为空栈
            stack.Clear();
            //循环处理表达式中的每一个字符
            for(int i=0;i<expression.Length;i++)
            {
                var ch=expression[i];
                if(ch=='(')
                    stack.Push('(');          //是左括号则压栈
                if(ch==')')
                {
                    //是右括号则弹栈
                    //如果栈是空的,说明没有与之匹配的左括号
                    if(stack.Count>0)
                        stack.Pop();
                    else
                        Console.WriteLine("括号不匹配");
                    return;
                }
            }
            if(stack.Count==0)
                Console.WriteLine("括号匹配");
            else
            Console.WriteLine("括号不匹配");
        }
    }
}
```

14.3.3 队列

和栈相类似,队列是一种特殊的线性表,它只允许在表的前端(front)进行删除操作,而在表的后端(rear)进行插入操作。进行插入操作的端称为队尾,进行删除操作的端称为队头。队列中没有元素时,称为空队列。在队列这种数据结构中,最先插入的元素将是

最先被删除的元素;最后插入的元素将是最后被删除的元素,因此队列又称为“先进先出”(first in first out,FIFO)的线性表。

队列可以用数组来存储,数组的上界即是队列所容许的最大容量。在队列的运算中需设两个索引下标:front,队头存放实际队头元素的前一个位置;rear,队尾存放实际队尾元素所在的位置。一般情况下,两个索引的初值设为 0,这时队列为空,没有元素。图 14-9 是一个队列的示意图。

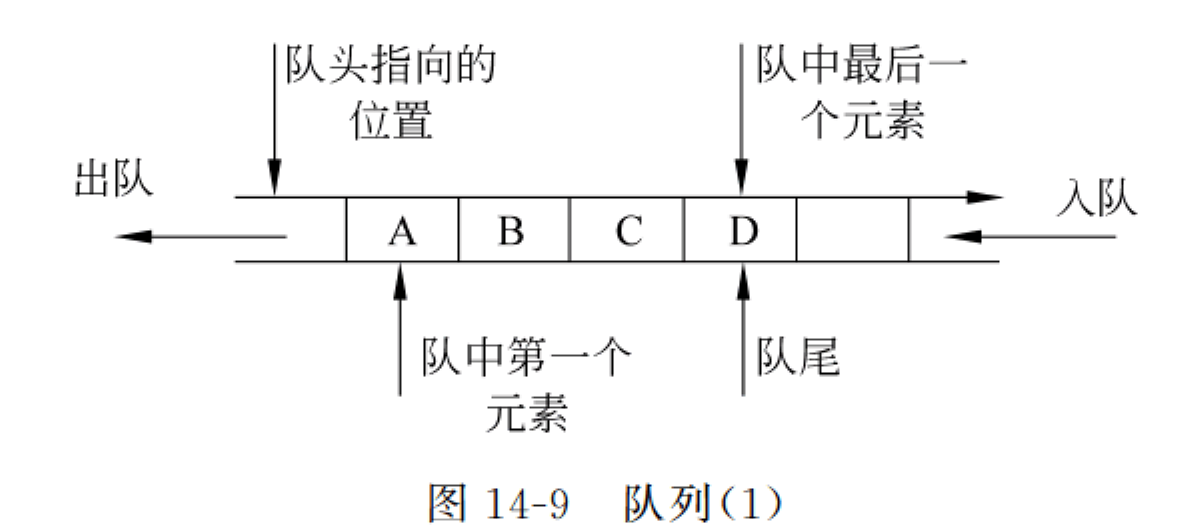


图 14-9　队列(1)

元素只能从队尾进入队列,从队头出队。也就是说要得到第 3 个元素 C,必须 A 和 B 先出队才可以。当队头和队尾相等时,表示队列是空的;队尾到达了数组的上界,则队是满的。图 14-10 是一个队列变化的示意图。图 14-10(a)是这个队列中有 2 个元素的情形;图 14-10(b)是当 A 和 B 出队后的情形,队列为空,此时队头等于队尾;图 14-10(c)是队中依次进入了 3 个元素 C、D 和 E;图 14-10(d)是 C 出队后队列的情形。

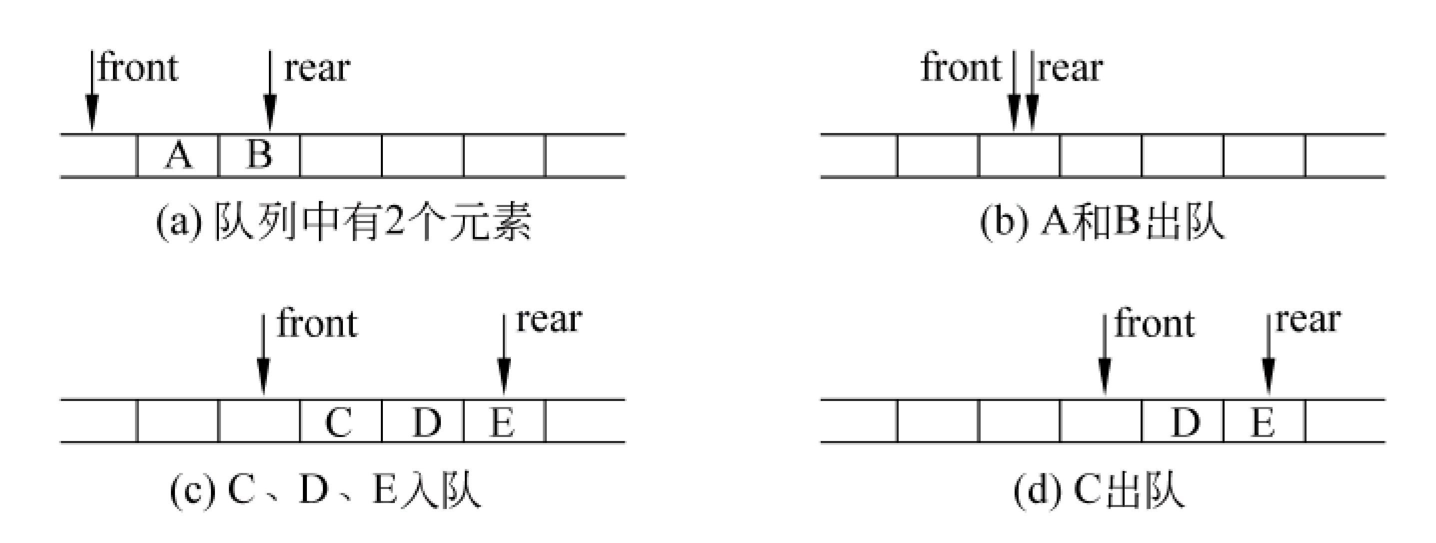


图 14-10　队列(2)

队列常用的操作有:

- 求队列的长度 GetLength,得到队列中数据元素的个数。
- 判断队列是否为空 IsEmpty,如果队列为空返回 true,否则返回 false。
- 清空队列 Clear,使队列为空。
- 入队 EnQueue,将新数据元素添加到队尾,队列发生变化。
- 出队 DlQueue,将队头元素从队列中取出,队列发生变化。
- 取队头元素 GetFront,返回队头元素的值,队列不发生变化。

仔细观察图 14-10 的过程,会发现随着元素的出队和入队,队头和队尾均会不断地向后移动。当队尾移动到整个队列存储空间的最后一个位置时,如果还有元素要入队,则会发生溢出。因为队尾已经移到最后,没法再向后移动了。但实际上,队列中还是有空间的,因为有元素出队,也就是说,队头之前的空间是可以再用来存储数据的。如何利用空间呢?最直观的方法是将队列整个向前移动,但这样做效率并不高。一个较好的办法是

将队列的头尾相连形成一个圆圈，空间可以反复利用。这就是所谓的循环队列。循环队列的示意图如图 14-11 所示。当队尾和队头重叠时，队列为空还是满呢？我们约定，当队头和队尾相等时，队空。当队尾加 1 后等于队头时，队满。这样虽然浪费了一个存储空间(为什么?)，但可以较为容易地区别队空和队满的情形。

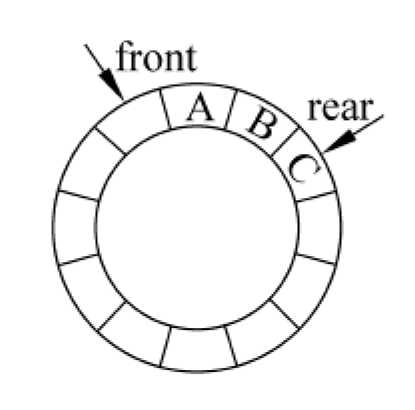
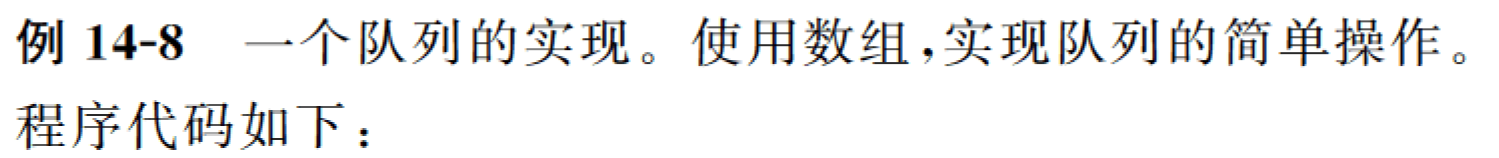


图 14-11　循环队列



例 14-8　一个队列的实现。使用数组，实现队列的简单操作。

程序代码如下：

```
using System;

namespace CSHARP14_8
{
    class Queue
    {
        private char[] q;
        private int front;
        private int rear;

        public Queue()
        {
            q=new char[100];
        }
        public Queue(int size)
        {
            q=new char[size];
        }
        ///<summary>
        //清空队列,也就是头尾均为 0
        ///</summary>
        public void Clear()
        {
            front=rear=0;
        }
        ///<summary>
        ///入队操作,为简单起见,没有判断队满的情形
        ///</summary>
        ///<param name="ch">入队的字符</param>
        public void EnQueue(char ch)
        {
            q[++rear]=ch;
        }
        ///<summary>
```

```
            ///出队操作,判断了队列是否为空
            ///</summary>
            ///<returns>出队的字符</returns>
            public char DIQueue()
            {
                if (!IsEmpty())
                    return q[++front];
                else
                    throw new Exception("Queue Empty!");
            }
            ///<summary>
            ///判断队列是否为空
            ///</summary>
            ///<returns>空位 true,非空为 false</returns>
            public bool IsEmpty()
            {
                return front==rear;
            }
            public int GetLength()
            {
                return rear - front;
            }
        }
}
```

测试代码如下：

```
using System;

namespace CSHARP14_8
{
        class Program
        {
            static void Main(string[] args)
            {
                Queue queue=new Queue();
                //清空队列
                queue.Clear();
                //入队操作,3个元素进入队列
                queue.EnQueue('A');
                queue.EnQueue('B');
                queue.EnQueue('C');
                //一个元素出队
                Console.WriteLine(queue.DIQueue());
                //出队 1个入队 3个后,全部出队
```

```
            Console.WriteLine(queue.DIQueue());
            queue.EnQueue('D');
            queue.EnQueue('E');
            queue.EnQueue('F');

            while(!queue.IsEmpty())
            {
                Console.Write(queue.DIQueue());
                Console.Write(' ');
            }
            Console.WriteLine();
        }
    }
}
```

运行结果：

```
A
B
C D E F
```

例 14-9 循环队列。使用数组，实现一个循环队列。这里的代码和例 14-8 基本是相同的，只是在入队、出队和打印等操作上注意对对头和队尾就队列的总长度取余。

程序代码如下：

```
using System;

namespace CSHARP14_9
{
    class MyQueue
    {
        private char[] q;
        private int front;
        private int rear;
        private int queueMax;            //队列最大容量

        public MyQueue()
        {
            q= new char[100];
            queueMax= 100;
        }
        public MyQueue(int size)
        {
            q= new char[size];
            queueMax= size;
        }
```

```
        public void Clear()
        {
            front= rear= 0;
        }
        public void EnQueue(char ch)
        {
            if((rear+ 1)%queueMax= = front)
            {
                throw new Exception("Queue Full!");
            }
            else
            {
                rear= (rear+ 1) %queueMax;
                q[rear]= ch;
            }
        }
        public char DeQueue()
        {
            if(!IsEmpty())
            {
                front= (front+ 1) %queueMax
                return q[ front];
            }
            else
            {
                throw new Exception("Queue Empty");
            }
        }
        public bool IsEmpty()
        {
            return rear= = front;
        }
        public int GetLength()
        {
            if (rear > front)
                return rear - front;
            else
                return queueMax+ rear - front;
        }
    }
}
```

14.3.4 Queue 类

和 Stack 类相似,Visual Basic 也提供了 Queue 类。该类实现了队列的常用算法,并

且当队列容量不足时会自动增加队列的大小。下面是一些主要的方法：

- Clear：从 Queue 中移除所有对象。
- Contains：确定某元素是否在 Queue 中。
- CopyTo：从指定数组索引开始将 Queue 元素复制到现有的一维数组中。
- Dequeue：移除并返回位于 Queue 开始处的对象。
- ToArray：将 Queue 元素复制到新数组中。
- Enqueue：将对象添加到 Queue 的结尾处。
- Peek：返回位于 Queue 开始处的对象但不将其移除。

和 Stack 类一样，Queue 类的唯一一个重要的属性是 Count，表示队列中含有元素的个数。

例 14-10 使用 Queue 类实现例 14-9。

程序代码如下：

```
using System;
using System.Collections.Generic;

namespace CSHARP14_10
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<char> queue=new Queue<char>();
            queue.Clear();
            //入队操作,3个元素进入队列
            queue.Enqueue('A');
            queue.Enqueue('B');
            queue.Enqueue('C');
            //打印队列
            PrintQueue(queue);
            //一个元素出队
            char ch=queue.Dequeue();
            PrintQueue(queue);
            //出队 1个入队 3个后,打印队列
            ch=queue.Dequeue();
            queue.Enqueue('D');
            queue.Enqueue('E');
            queue.Enqueue('F');
            PrintQueue(queue);
        }
        private static void PrintQueue(Queue<char> q)
        {
            char[] qArray=new char[q.Count];
```

```
                //先将队列中的所有元素复制到一个数组中
                q.CopyTo(qArray,0);
                foreach(var ch in qArray)
                    Console.Write("{0} ",ch);
                Console.WriteLine("队列的长度是 {0} ",q.Count);
            }
        }
}
```

14.4　图和树

前几节讲述了一些线性结构的数据，而图(graph)和树则是非线性的数据结构。同时，现实中的很多问题也是用线性数据结构无法描述的，需要借助非线性的数据结构来描述。

14.4.1　图的基本概念

1736 年，数学家欧拉(Euler)发表了著名的论文《柯尼斯堡七座桥》。在该论文中，首先使用图的方法解决了柯尼斯堡七桥问题，从而欧拉也被誉为图论之父。这个问题是基于一个现实生活中的实例：当时东普鲁士柯尼斯堡(Königsberg，今日俄罗斯加里宁格勒)市区跨普列戈利亚河(Pregel)两岸，河中心有两个小岛。小岛与河的两岸有 7 座桥连接。于是，7 座桥将 4 块陆地连接了起来，如图 14-12 所示。城里的居民想在散步的时候从任何一块陆地出发，经过每座桥一次且仅经过一次最后返回原来的出发点。当地的居民和游客做了不少尝试，却都没有成功；而欧拉最终解决了这个问题，并断言这样的回路是不存在的。

欧拉在解决问题时，用 4 个结点来表示陆地 A、B、C 和 D。凡是陆地间有桥连接的，便在两点间连一条线，于是图 14-12 转换为图 14-13。

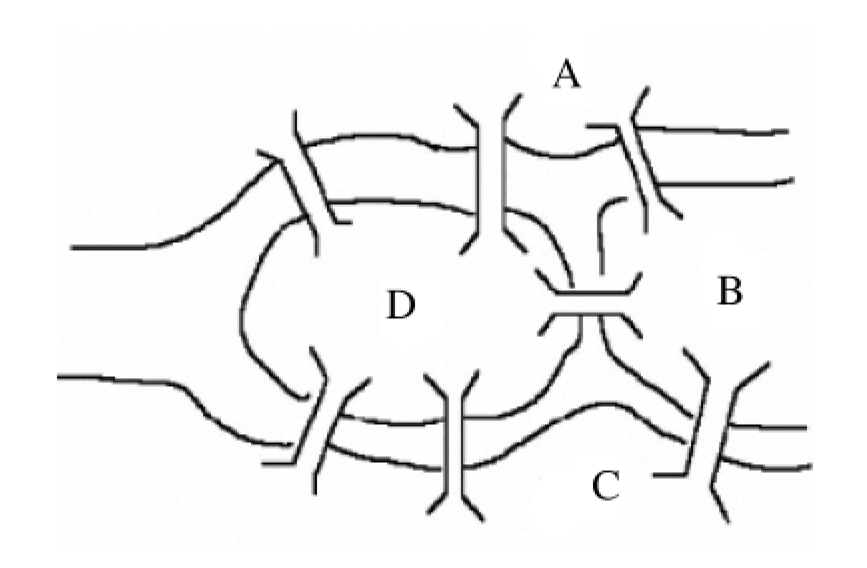


图 14-12　柯尼斯堡七桥问题示意图

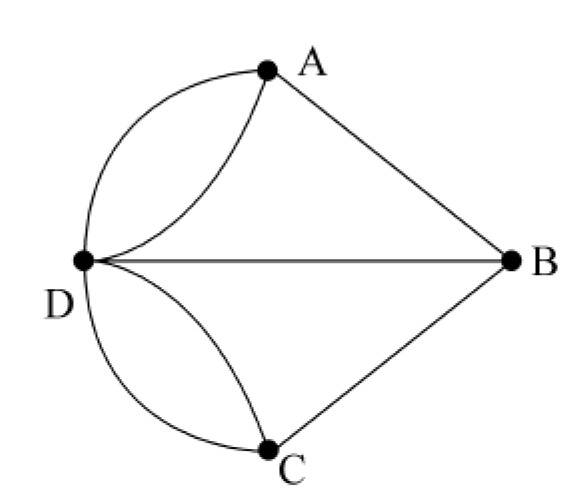


图 14-13　柯尼斯堡七桥问题抽象后的表示

此时，问题转化为从图 14-13 中的 A、B、C、D 任一点出发，通过每条边一次且仅一次后回到原出发点的回路是否存在？欧拉断言了这个回路是不存在的，理由是从图 14-13 中的任一点出发，为了能够回到原出发点，就要求与每个点关联的边数均为偶数。这样才能保证从一条边进入某点后可以从另外一条边出来。而图 14-12 中的 A、B、C、D 全部都

与奇数边关联，因此回路是不存在的。

由上面的例子我们也看到，所谓图是由结点或称顶点（vertex）和连接结点的边（edge）所构成的图形。使用 V(G)表示图 G 中所有结点的集合，E(G)表示图 G 中所有边的集合。则图 G 可记为＜V(G)，E(G)＞或＜V，E＞。有 n 个顶点和 m 条边的图记为(n，m)图或称为 n 阶图。

例 14-11 有 4 个城市 v_1、v_2、v_3 和 v_4。v_1 和其他 3 个城市都有道路连接，v_2 和 v_3 之间有道路连接，画出图并用集合表示该图。

显然结点集合 $V=\{v_1,v_2,v_3,v_4\}$，边集合 E＝{v_1 和 v_2 之间的边，v_1 和 v_3 之间的边，v_1 和 v_4 之间的边，v_2 和 v_3 之间的边}。画出的图如图 14-14 所示。

更一般地，边可以用结点对来表示：

$$V=\{v_1,v_2,v_3,v_4\}$$

$$E=\{(v_1,v_2),(v_1,v_3),(v_1,v_4),(v_2,v_3)\}$$

在图中，如果边不区分起点和终点，则这样的边称为无向边。所有边都是无向边的图称为无向图，如图 14-14 就是一个无向图。反之，若边区分起点和终点，则为有向边，所有边都是有向边的图称为有向图。在图中，有向边使用带有箭头的线段表示，有起点指向终点。在集合中则用有序对＜v_1，v_2＞来表示，图 14-15 是一个有向图的示例。

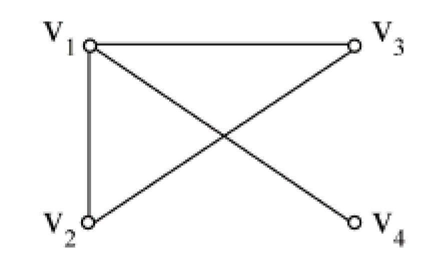


图 14-14 例 14-11 中的图

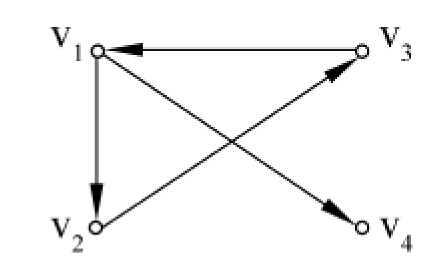


图 14-15 一个有向图的示例

在图 14-15 中：

$$V=\{v_1,v_2,v_3,v_4\}$$

$$E=\{\langle v_1,v_2\rangle,\langle v_1,v_4\rangle,\langle v_3,v_1\rangle,\langle v_2,v_3\rangle\}$$

结点的度是指和结点关联的边的个数。如在图 14-14 中，v_1 的度是 3，v_2 和 v_3 的度是 2，v_4 的度是 1。对于有向图，则区分为出度和入度。由结点指向外的边的个数为出度，反之为入度。如图 14-15 中，v_1 的出度为 2，入度为 1；v_4 的出度为 0，入度为 1。

图在计算机中如何存储，是人们普遍关心的一个问题。简单的方法是将图用一个二维矩阵来表示，这样的矩阵通常称为邻接矩阵。在此我们不系统讨论，仅以图 14-15 的存储为例来说明。

例 14-12 将简单有向图（见图 14-15）以邻接矩阵的方式存储到计算机中。

要以邻接矩阵的方式存储，首先需要对结点指定一个次序。在此，就以结点的下标从小到大为序，排列为 v_1,v_2,v_3,v_4。然后使用一个 4×4 的矩阵来存储该图，矩阵中的元素只有 2 个取值：0 或者 1。对于 2 个结点 v_i 和 v_j，若 v_i 和 v_j 之间存在一条边，则对应的矩阵元素 $a_{ij}=1$，反之则为 0。图 14-15 示例的有向图存储矩阵如图 14-16 所示。

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

图 14-16 存储的邻接矩阵

容易看出，矩阵中 1 的个数对应图中边的个数，而对角线

的元素则全为 0。

14.4.2 带权图和最短路径

图的问题异常复杂，甚至是一门完整的学科——图论。在此无法对图有一个完整系统的讨论。为了使读者对图有进一步的认识，作为一个例子，简单介绍带权图及最短路径的算法，并以此结束对图的讨论。

在处理有关图的实际问题时，往往有值的存在，比如距离、运费、城市、人口数以及电话部数等。一般这个值称为权值。在图中，将每条边都有一个非负实数对应的图称为带权图或赋权图。这个实数称为这条边的权。根据不同的实际情况，权数的含义可以各不相同。例如，可用权数代表两地之间的实际距离或行车时间，也可用权数代表某工序所需的加工时间等。如图 14-17 便是一个带权图。

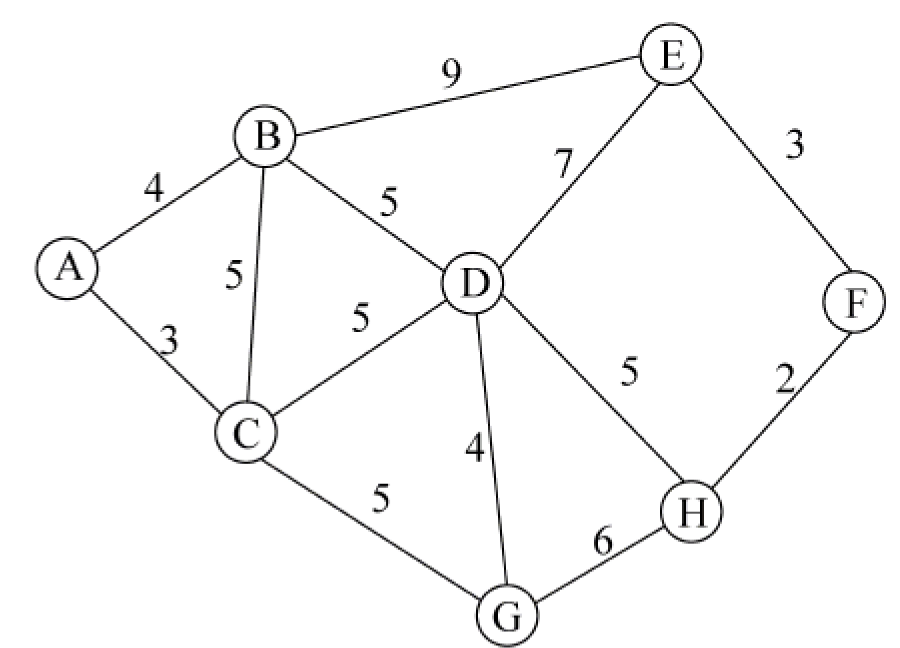


图 14-17 带权图

对图 14-17 所示的无向带权图求最短路径是一个经常遇到的很实际的问题。假设在图中的 A 到 G 点表示 8 个村庄，边表示村庄之间的道路。边上的权值表示距离。现在的问题是从 A 到 F 最短的距离是多少？

求最短路径的算法是 E. W. Dijkstra 于 1959 年提出来的，这是至今公认的求最短路径的最好方法，我们称它 Dijkstra 算法。假定给定带权图 G，要求 G 中从 v_0 到 v 的最短路径，Dijkstra 算法的基本思想是：

将图 G 中结点集合 V 分成两部分：一部分称为具有 P 标号的集合，另一部分称为具有 T 标号的集合。所谓结点 a 的 P 标号是指从 v_0 到 a 的最短路径的路长；而结点 b 的 T 标号是指从 v_0 到 b 的某条路径的长度。Dijkstra 算法中首先将 v_0 取为 P 标号结点，其余的结点均为 T 标号结点，然后逐步将具有 T 标号的结点改为 P 标号结点，当目的结点也被改为 P 标号时，则找到了从 v_0 到 v 的一条最短路径。下面通过一个例子给出实际的算法步骤。

例 14-13 计算图 14-16 所示的带权图中，从 A 点到 F 点的最短路径。

(1) 首先，将起点 A 划归为 P 标号集合，其余的结点均为 T 结点。A 到 A 的距离为 0，所以 A 的 P 标号为 0。

(2) 更新 T 中结点到 A 的距离。如该结点和 A 相邻(有边连接)，则距离就是边的权值；如和 A 没有直接的边连接，则距离是无穷大。

(3) 在 T 中找到一个值最小的结点，并将其划归到 P 集合。此时，计算的结果如图 14-18 所示(C 结点进入 P 集合)。

(4) 根据新进入的 C 结点，更新与 C 相连的结点的值。新值等于 C 的 P 结点值加上到与其相连的结点的距离(边的权值)。更新的算法是：如果新值小于原有的值，则用新的值取代，否则保持原有值不变。

(5) 重复步骤(3)和(4)，直到目标点进入 P 集合。

图 14-18～图 14-24 演示了这一过程。

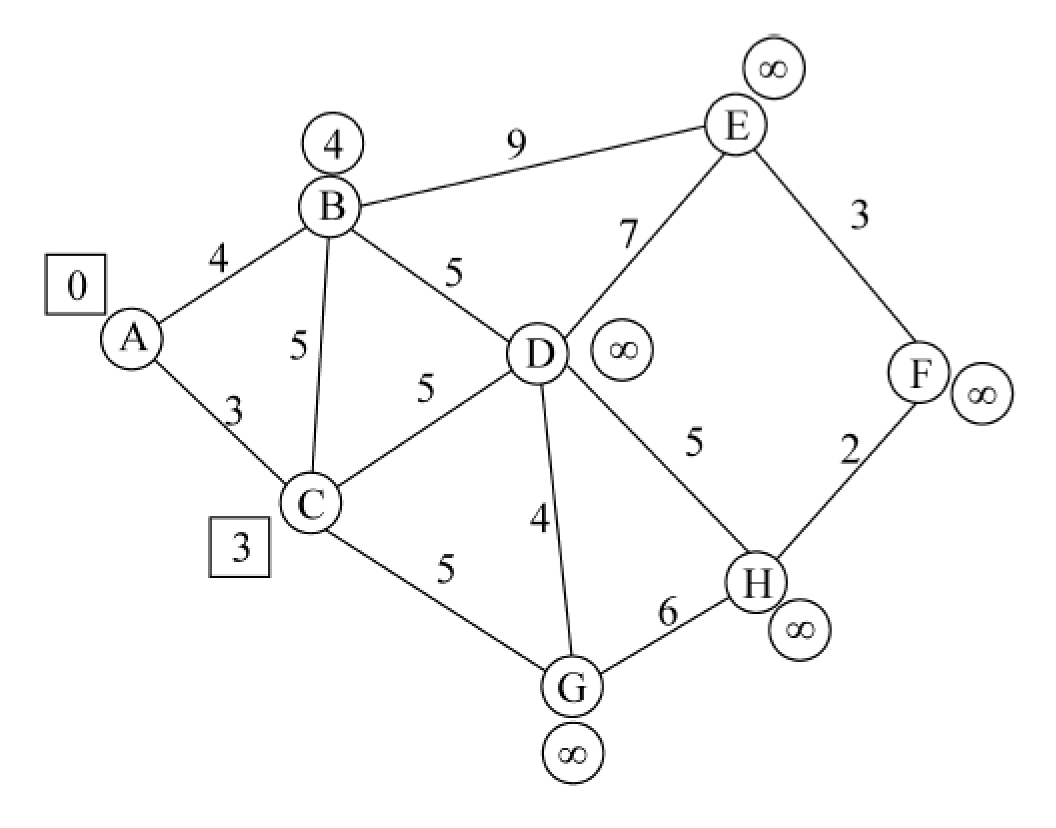


图 14-18　最短路径(1)

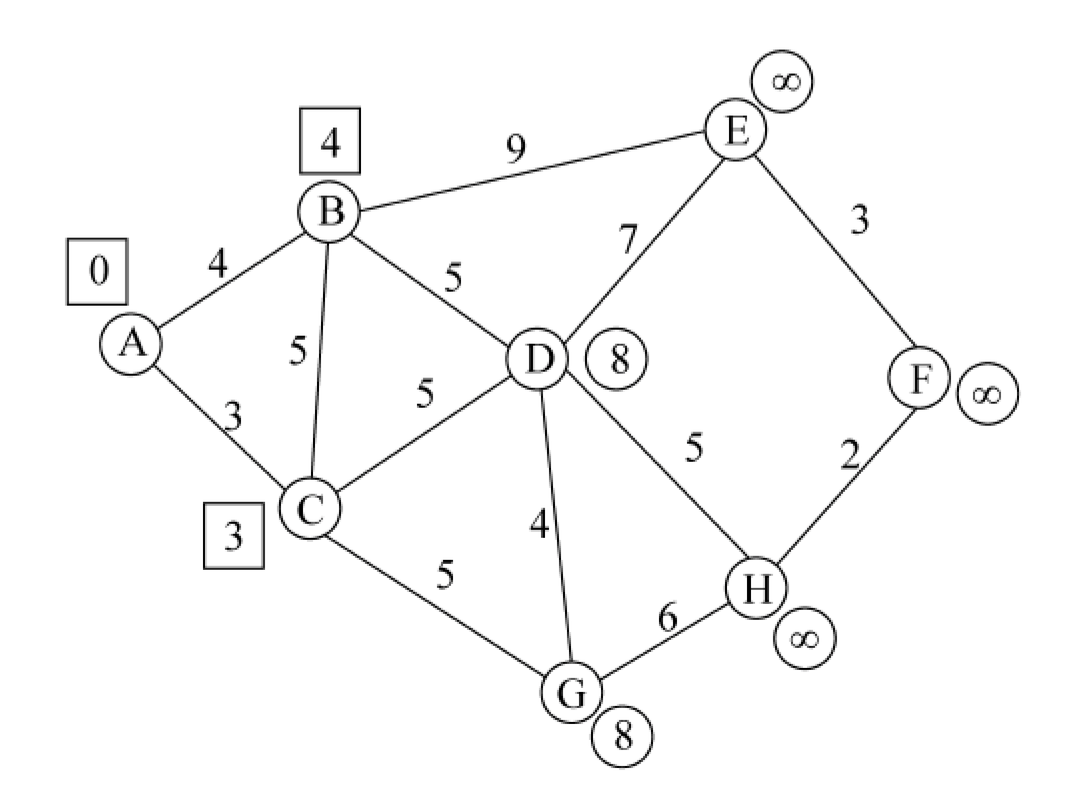


图 14-19　最短路径(2)

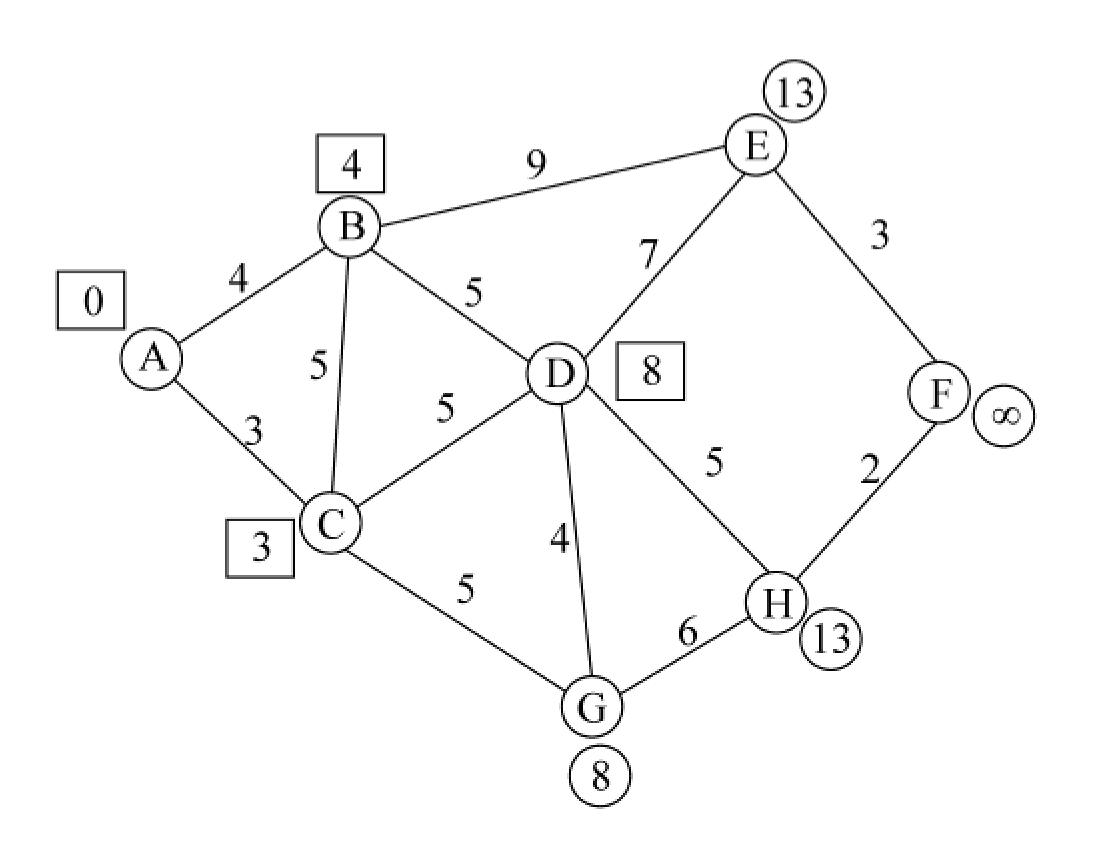


图 14-20　最短路径(3)

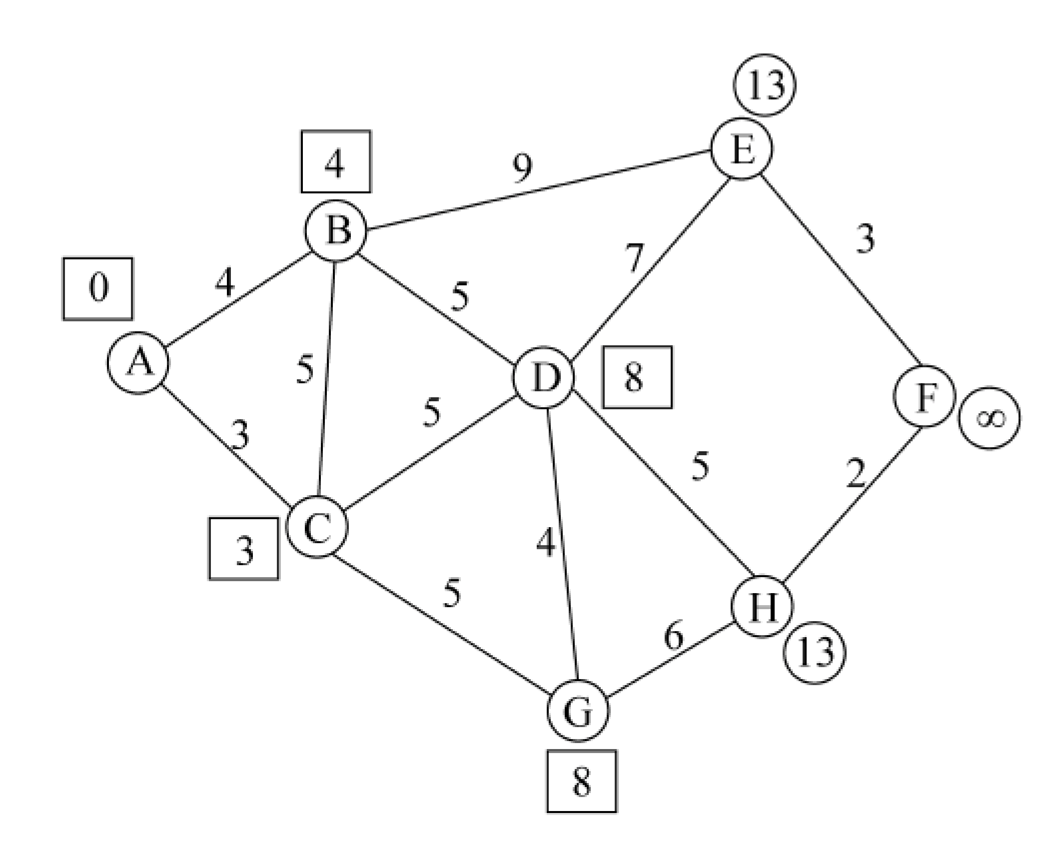


图 14-21　最短路径(4)

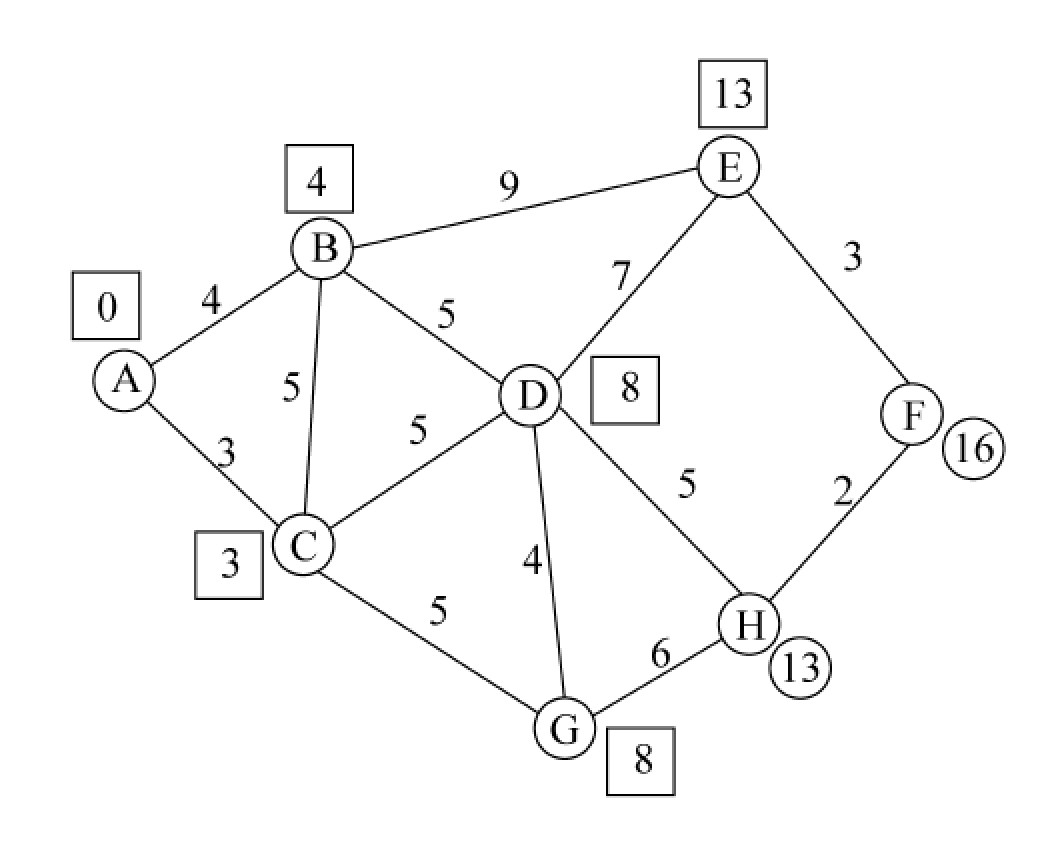


图 14-22　最短路径(5)

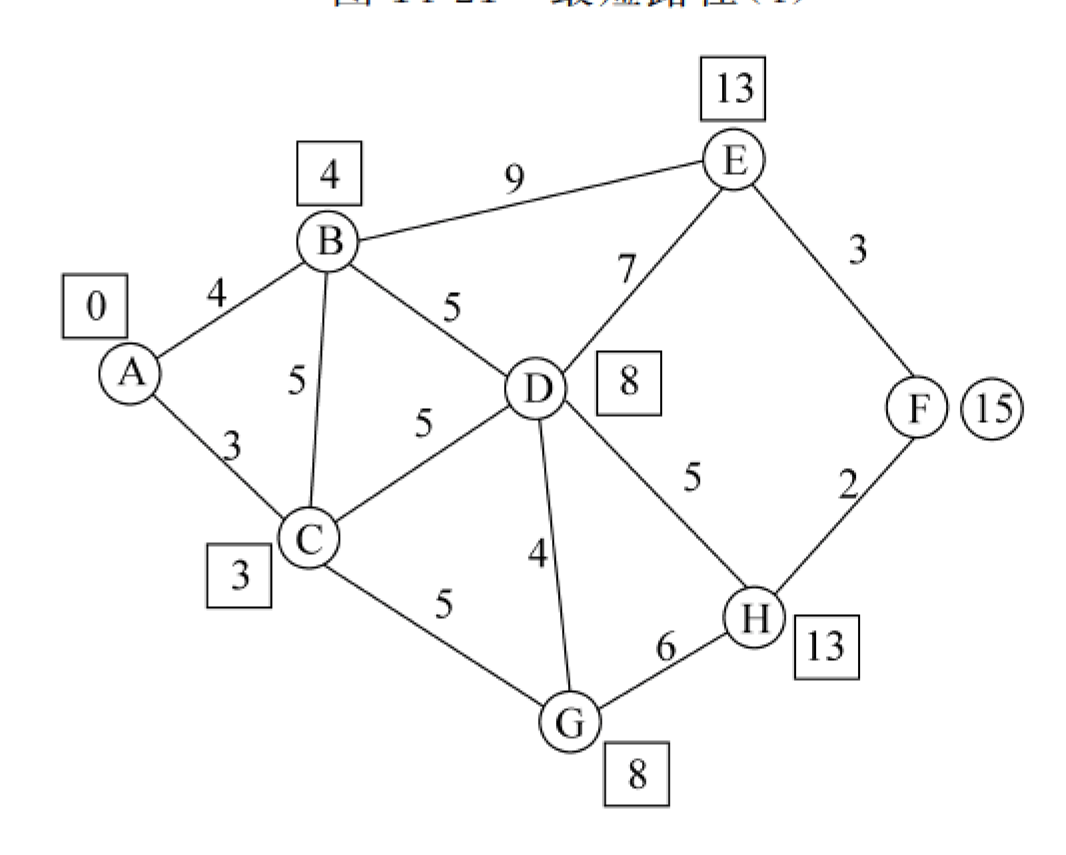


图 14-23　最短路径(6)

在图 14-18 中，A 到 B 的距离为 4，到 C 的距离为 3，到其余结点的距离为无穷大。由于 C 结点的值最小，因此 C 进入 P 集合(P 集合以方框表示，T 集合用圆圈表示)。

在图 14-19 中，结点 C 进入 P 集合后，到 B 的距离为 3＋5＝8，大于 B 原来的 4，因此

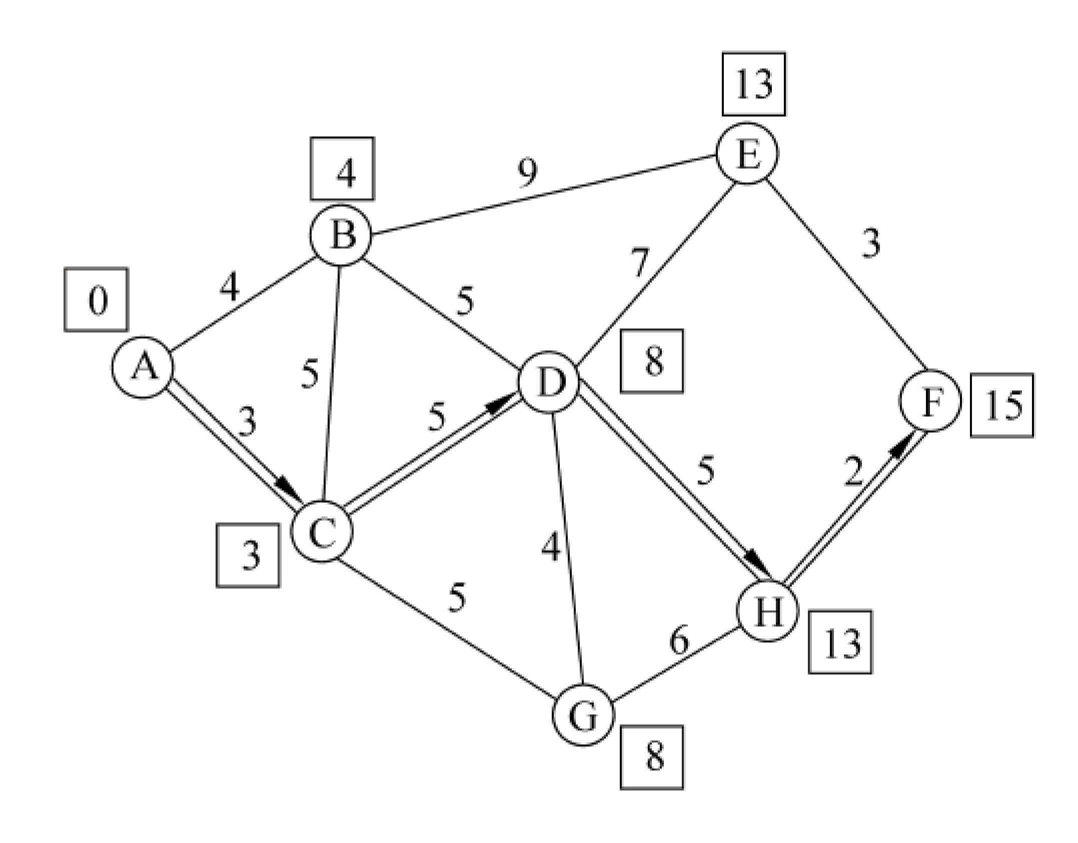


图 14-24 最短路径(7)

B 的值不变。而到 D 和 G 的值均为 8,均小于原来的无穷大,因此用 8 取代原来的值。之后,在 T 中,B 的值为 4 最小,B 进入 P 集合。

在图 14-20 中,结点 B 进入后更新与 B 连接的 D 和 E 值。其中 D 的值不变,E 为 13。此时 D 和 G 均有最小值 8,任取一个进入 P,在此取的是 D。然后又更新了 H 的值。G 的原值小于 8+4,因此保持不变。

在图 14-21 中,结点 G 的值最小进入 P,H 的值未变。

在图 14-22 中,任选 E 进入 P,F 值变为 16。

在图 14-23 中,结点 H 进入 P,F 的值变为 15。

在图 14-24 中,终点 F 进入 P 集合,运算结束。从 A 到 F 的最短距离为 15。而事实上,对于每一个 P 中的结点值,计算出了从 A 到该结点的最短距离,如到 E 的最短距离为 13。而找到最短路径的方法是用 F 点的 P 值减去边的权值,倒推回 A 点。如 F 的值 15−2=13 和 H 吻合,而不是 E(因为 15−3=12 不等于 E 的 13)。

14.4.3 树的基本概念

树可以看作是一个特殊的有向图。对于一个有向图,如果

(1) 存在一个特殊的结点 r,其入度等于 0。

(2) 除了 r 外的其他结点的入度均为 1。

(3) R 到图中其他结点均有路可达。

则满足这样的图称为树。其中入度为 0 的结点称为根,出度为 0 的结点称为叶子,出度不为 0 的结点称为分枝点,如图 14-25 所示。

在画树的图的时候,由于所有的箭头方向都是一致的,所以箭头常常省略,如图 14-25 所示。树是有层次的,层次指的是从根到该结点的距离。称距根最远的叶子的层数为树的高度。图 14-26 的树的高度为 3。同一层次之间的结点称为兄弟,上一层次的称为父亲,下一层次的称为儿子,如图 14-26 中对 h 结点的描述。

对于一棵树而言,若所有结点的入度均小于等于 m,则称此树为 m 叉树。如果每个结点的入度都相等且都等于 m,则称此树为完全 m 叉树。在计算机学科中经常应用的是二叉树。

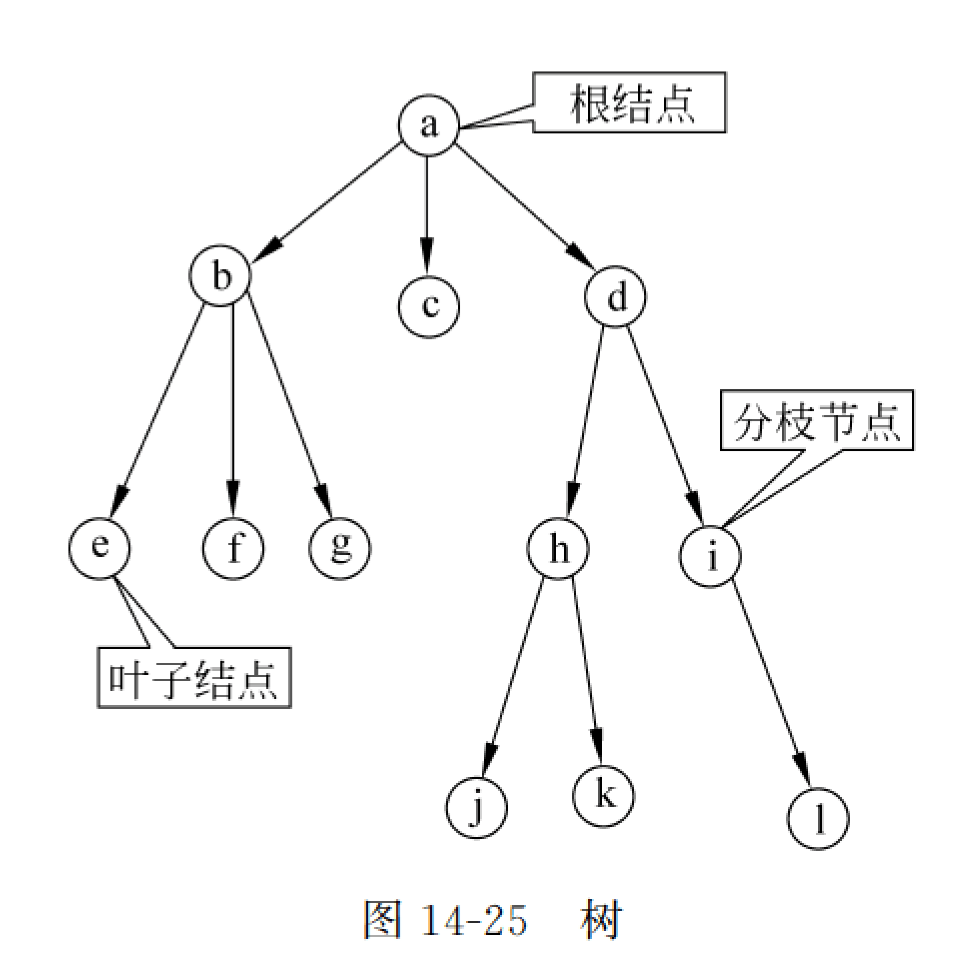


图 14-25　树

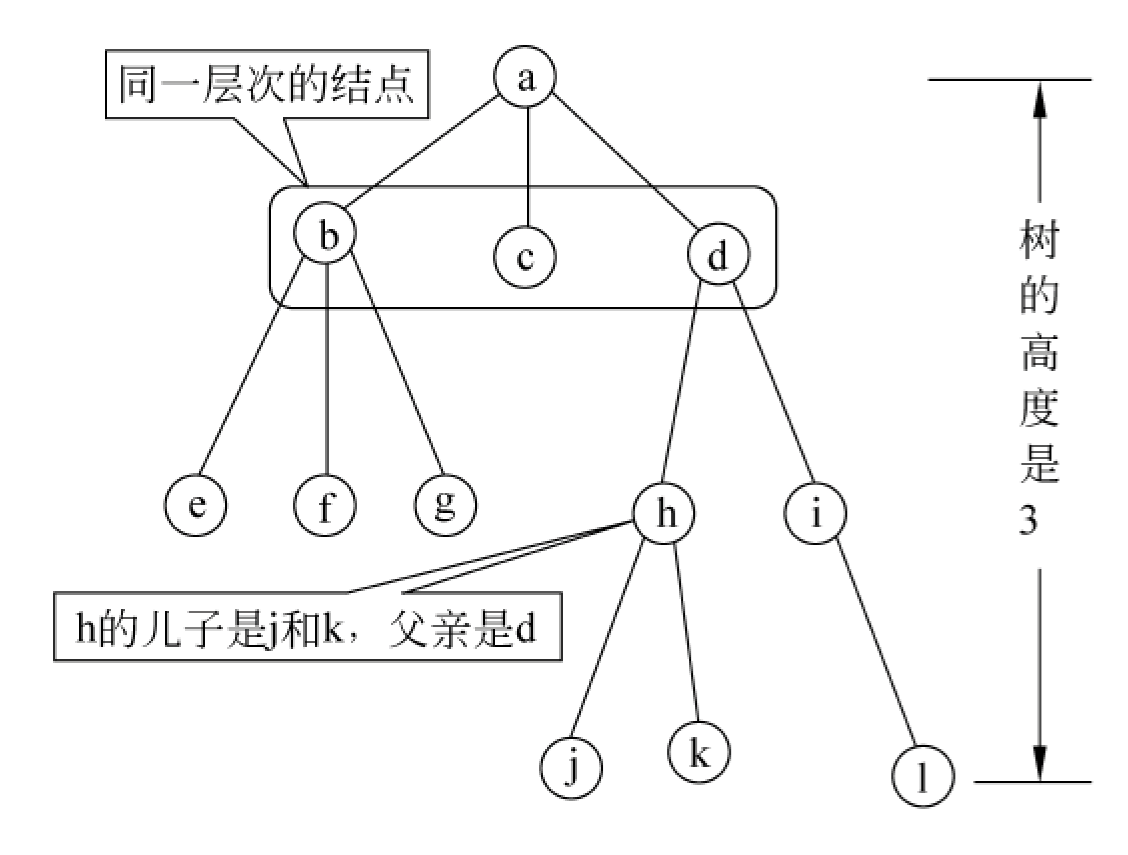


图 14-26　树的高度及层次关系

14.4.4　二叉树

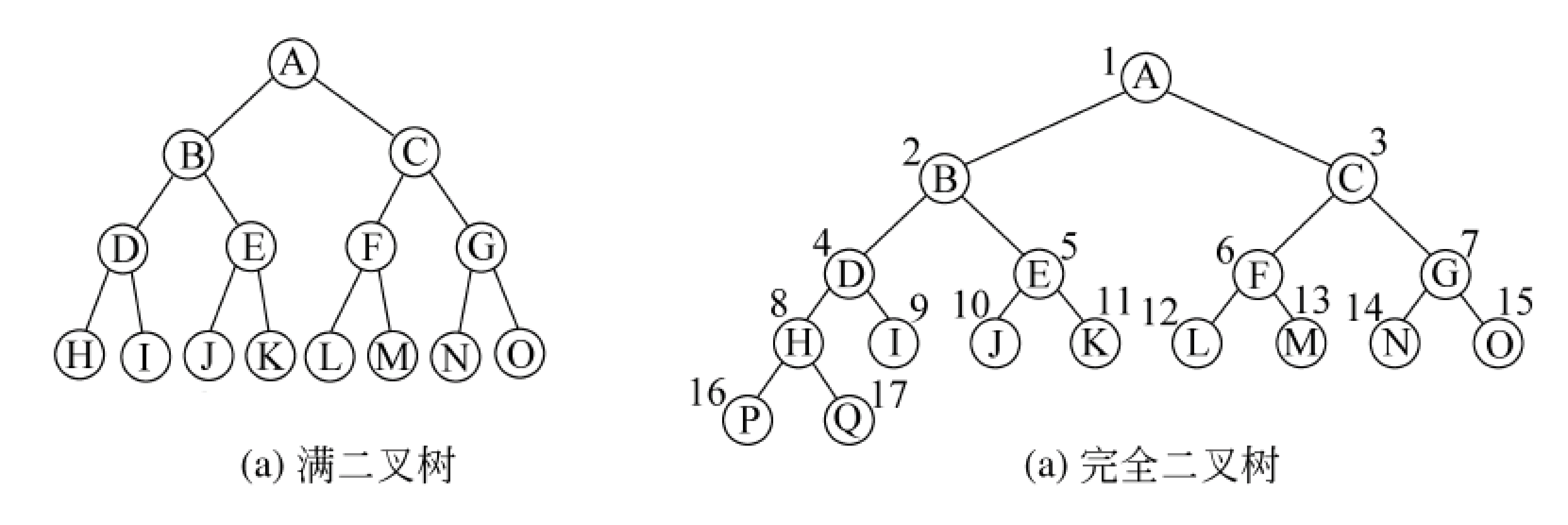
二叉树是每个结点最多有两个子树的树结构。通常子树被称作“左子树”(Left Subtree)和“右子树”(Right Subtree)。二叉树的每个结点至多只有两棵子树(不存在出度大于 2 的结点)。二叉树的子树有左右之分,次序不能颠倒。

二叉树具有如下性质:

- 二叉树的第 i 层至多有 2^{i-1}个结点。
- 深度为 k 的二叉树至多有 2^{k-1}个结点。
- 二叉树的结点个数可以为 0。
- 二叉树的结点有左、右之分。

一棵深度为 k,且有 2^{k-1}个结点的二叉树,称为满二叉树(Full Binary Tree)。这种树的特点是每一层上的结点数都是最大结点数。

而对于深度为 k,有 n 个结点的二叉树,当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时,称之为完全二叉树(Complete Binary Tree)。也就是说,若一棵二叉树至多只有最下面的两层上的结点的度数小于 2,并且最下层上的结点都集中在该层最左边的若干位置上,则此二叉树称为完全二叉树。具有 n 个结点的完全二叉树的深度为 $\log_2 n+1$。深度为 k 的完全二叉树,至少有 2^{k-1}个结点,至多有 2^k-1个结点。图 14-27(a)、(b)分别是满二叉树和完全二叉树的示例。



(a) 满二叉树　　(a) 完全二叉树

图 14-27　满二叉树和完全二叉树

14.4.5　树的遍历

树的遍历是树的一种重要的运算。所谓遍历是指对树中所有结点的信息的访问，即依次对树中每个结点访问一次且仅访问一次。树的遍历有两种方式：先根遍历，即先访问树的根结点，然后依次先根访问根的每一颗子树；后根遍历，即依次后根访问根的每一颗子树，最后访问根结点。

对于图 14-26 的树，采用先根方式，结点访问的次序依次为：

a b e f g c d h j k i l

采用后根方式，结点访问的次序依次为：

e f g b c j k h l i d a

习题

1. 什么是数据的线性存储结构？什么是数据的非线性存储结构？

2. 简述线性表的操作。

3. 假设电话号码簿由人名和一个电话号码组成。设计一个线性表，存储 7 个人的电话号码簿。

4. 设栈 S 中存储的是字符数据，自栈底到栈顶依次为 A、C、D。经过两次出栈操作并将 E 压入栈，此时栈中的数据是什么？

5. 使用栈，检查表达式(2＋3) * a * (3＋b)/(2 * (12＋8)的括号是否匹配。

6. 编写程序，输入一行文本，然后使用栈逆序显示该行文本。

7. 编写程序，用栈来判断一个字符串是否为回文(即顺读和倒读都相同的字符串)。程序忽略字符串中的大小写、空格和标点符号。

8. 使用 LinkedList 类实现第 3 题的电话号码簿。打印该电话号码簿，然后删去第 2 个和最后一个结点的数据再次打印该电话号码簿。

9. 设计一个队列，将整数 3，4，5 进入队列，打印该队列；将队列的前两个元素出队，随后将 11 和 12 入队，再次打印该队列。

10. 对于图 14-11 的循环队列，在该图的基础上，将 1、2、3、4、5 入队，并将两个元素出队后，画出队列目前的状态。

11. 将图 14-17 的带权图使用邻接矩阵的方式存储到计算机中，试写出该矩阵。(提示：该矩阵是一个对角线元素为 0 的对称矩阵。)

12. * 试描述使用邻接矩阵(在第 11 题的基础上)计算最短路径时的算法。

13. * 试编写程序(在第 11 题和第 12 题的基础上)实现图 14-23 最短路径的计算。

参 考 文 献

1. Paul Deitel,等. 张思宇,等译. Visual C# 2010 大学教程. 4 版. 北京：电子工业出版社,2011.
2. Microsoft MSDN. https://msdn. microsoft. com/zh-cn.
3. Karli Watson,Christian Nagel. 齐立波译. C#入门经典. 5 版. 北京：清华大学出版社,2010.